THE Smalltalk REPORT

SIGS PUBLICATIONS

Publishers of JOURNAL OF OBJECT-ORIENTED PROGRAMMING, OBJECT MAGAZINE, C++ REPORT, SMALLTALK REPORT, THE X JOURNAL, REPORT ON OBJECT ANALYSIS & DESIGN, OBJECTS IN EUROPE, DIRECTORY OF OBJECT TECHNOLOGY, and OBJEKT SPEKTRUM (Germany)

Table of Contents

October 1994 **Vol4 No2**

Features

Columns

Departments

The Smalltalk Report (ISSN# 1056-7976) is published 9 times a year, monthly except in Mar–Apr, July–Aug, and Nov–Dec. Published by SIGS Publications Inc., 71 West 23rd St., 3rd Floor, New York, NY 10010. Mailed First Class. Canada Post International Publications Mail Product Sales Agreement No. 290388.
Subscription rates 1 year (9 issues): domestic, $79; Foreign and Canada, $114. To submit articles, please send electronic files on disk to the Editors at 885 Meadowlands Drive #509, Ottawa, Ontario K2C 3N2, Canada, or via Internet to streport@objectpeople.on.ca. Preferred formats for figures are Mac or DOS EPS, TIF, or GIF formats. Always send a paper copy of your manuscript, including camera-ready copies of your figures (laser output is fine).
POSTMASTER: Send address changes and subscription orders to: The Smalltalk Report, P.O. Box 2027, Langhorne, PA 19047. For service on current subscriptions call 215.785.5998, 215.785.6073 (fax), P00976@psilink.com (email). PRINTED IN THE UNITED STATES.

Editors' Corner

JOHN PUGH

PAUL WHITE

We have been writing over the past few issues about what has been happening in the Smalltalk marketplace and community, discussing the activities in new products and the plethora of conferences that are dealing with the issue of Smalltalk. The one area that we have for the most part neglected over the last while is the role played by what we call hard-core Smalltalk users, or the "Smalltalk gurus" that are out there. Smalltalk allows for that unique echelon of programmer to really shine, which is always a pleasure to watch. In fact, we have the pleasure to work with a few people who have this innate ability to make code appear on the screen that just works and yet is built in such an elegant fashion—it's like watching a maestro conducting a symphony. It's not that all languages don't have their gurus, it's just that unlike most languages where gurus often generate code quickly but which is often difficult for others to understand, Smalltalk gurus tend to generate things that appear to be obvious approaches—they make it very understandable for all of us.

It is difficult to isolate what techniques are being employed by excellent Smalltalk programmers that allow them to write such elegant code. We are constantly asked for good rules of thumb when it comes to programming techniques that will allow people to best utilize the powers offered by Smalltalk. While it's obvious no one has the definitive answer to this, here are a few observations we've made.

1. Code walkthroughs are a must, and they must have bite. Everyone must push each other to do a better job.
2. While it seems a trivial point, choosing meaningful names saves an enormous amount of browsing time. And don't allow abbreviations, they make using a library very difficult. What's more, what inevitably gets abbreviated away is whether something is singular or plural and it's this little piece of information that makes all the difference in the world. Finally on this theme, why is it so many people are choosing to add prefixes and suffixes to their class names, and even their method names? While recognizing the name space problem that exists in Smalltalk, it makes a library more difficult to use.
3. Complete the library. Don't just write for yourself, but think of how others may use the library in the future. Two simple examples that have always struck us are "why the class Date has a method "today," but not "tomorrow" or "yesterday." The other is the problem of different possible spellings for the same message. Why, for example, is the method "sqrt" in the library, but not "squareRoot"?
4. Consolidate. Good Smalltalkers will tell you they throw away up to half of what they write — this has to be seen as a healthy thing and encouraged.
5. Allow play time. The best way we've seen to understand Smalltalk is to build some tools for it. The skills to be gained are far more important than the tools themselves.

Related to this topic, an issue that we have raised in passing a few times, which we have never thoroughly seen addressed, is the issue of project management for Smalltalk projects. We all know the development process is drastically different because of the flexibility and productivity possible with Smalltalk. But to managers who are new to this whole process, this can pose a frightening prospect. Just how does one know how far along a project is? Is it on budget? Are there still "gotchas" left in a project that haven't been fleshed out yet? We need managers with real experience to weigh in on this topic.

And finally, just a quick note about the OOPSLA conference being held this month in Portland. As many of you are aware, OOPSLA is the main object-oriented conference and was instrumental in making objects, and Smalltalk in particular, a mainstream development language. We certainly have a fond spot for the conference that has traditionally been different from all the other conferences in that it has always had a very strong Smalltalk presence and highly technical personnel staffing the exhibit booths. For those of you going to the conference, drop by and say hello.

Enjoy the issue.

STEP INTO THE FUTURE WITH THE COMPANY THAT DEFINED OBJECT TECHNOLOGY SERVICES

When object oriented programming was in its infancy, Knowledge Systems Corporation was already putting it to work in companies like yours. Today, we're positioned to take you into the future of object technology in ways that no other company can. With the most complete range of services in the industry, KSC can assure your successful object transition every step of the way. Classroom instruction, project-focused apprenticeships, and consulting are all part of our exclusive commitment to object technology services.

Once you've made the decision to move to object technology, you want to get the benefits as quickly as possible. KSC offers a complete curriculum of classroom education, at your site or in our corporate training facility. These courses help you establish a firm foundation in object technology concepts and Smalltalk programming.

To cut months off your transition time, we've developed an exclusive Smalltalk Apprentice Program (STAP). Already proven in companies such as American Management Systems, GE Capital Corporation, IBM, Northern Telecom, The Prudential, Southern California Edison and Sprint, the STAP is a total immersion, project-focused program that compresses six to ten months of learning experience into four to six weeks.

KSC can also tackle your object technology projects head-on with the most experienced analysts, designers and programmers in the business. You can outsource the entire job, or use our consultants to lend expertise to your own development group.

In addition to our service offerings, KSC is a distributor of third party tools such as ENVY®/Developer, the premier Smalltalk team development environment.

If you're ready to step into the future of object technology, call the one company that will lead you there—Knowledge Systems Corporation, 919-481-4000. Or email: salesinfo@ksccary.com. 4001 Weston Parkway, Cary, North Carolina 27513.

KNOWLEDGE SYSTEMS CORPORATION

919-481-4000

ENVY is a registered trademark of Object Technology International Inc.

Persistent object management using the ParcPlace Binary Object Streaming Service

Michael Christiansen

Moving information between Smalltalk images can be accomplished using the fileIn: / fileOut: and storeOn: / readFrom: operations provided by Object, Class, and other support classes. Information is written in ASCII format as Smalltalk expressions that, when read and evaluated in through a fileIn: or readFrom: operation, recreates the original classes and instances in the target image.

Although these operations work fine for moving source code between images, they are not well suited for moving arbitrary structures of objects. There are problems with this approach.[1,2] These problems include:

- The storeOn: mechanism cannot deal with circular references. Without special assistance the operation can get caught in an infinite loop when two objects mutually reference each other.
- To read an expression requires the presence of the Smalltalk compiler, which is often stripped from commercial releases of a product.
- No facilities are provided for controlling the depth of the copy of arbitrary object structures being copied to the file.
- ASCII representations are bulky and slow to read and write from the file.

ParcPlace ObjectWorks and VisualWorks provide the Binary Object Streaming Service (BOSS), which allows one or more structures of arbitrary objects to be stored and later retrieved from a file.

The materials described in this article were developed during the development of Cellular Performance Management System (CPMS), a cellular network planning, configuration, and performance management system. We have applied BOSS as a temporary solution to our persistent storage needs until such time as a true object-oriented DBMS can be selected and integrated. This article is intended as a tutorial in the use of this facility as described in the ParcPlace manuals, and includes our experiences and the problems we have addressed during this effort.

BINARY OBJECT STREAMING SERVICE

BOSS is a facility provided by ParcPlace Smalltalk that allows arbitrary structures of objects to be written to, and later retrieved from, a file stream. Objects are stored in an internal binary format that is more compact and faster to retrieve than is the use of expressions described above.

Generally, BOSS allows the user to write a representation of an object into an archive that includes the object's class identifier and its instance variables. These variables, themselves objects, are represented and stored. This process is repeated in a depth-first transversal until the entire tree, rooted with the original *root object*, has been stored in the archive file.

This process of transversing, or *tracing*, the members of a single root object and storing each in turn in a BOSS archive is similar to making a deep copy of the object. But the depth-first transversal of the root object is bounded by certain conditions under which the further tracing of an encountered object is halted. For example, if an encountered object being traced is an immediate value (an integer, string, etc.) its value is stored and the trace goes no further. Also, if an encountered object has already been traced and stored in the archive, a reference to the stored object is archived in place of the object in its current context.

There are classes whose instances receive special treatment when encountered during the tracing operation. These classes include all class objects and certain global variables. For example, Smalltalk and Processor are recorded symbolically in an archive when encountered during a trace.

Another class of objects that cannot be placed into a BOSS archive are those whose contents only have meaning within the image being executed and that cannot be exported. Examples of these context-sensitive objects include OSHandle, GraphicsHandle, and other objects whose states are tied to the executing image and to the virtual machine's interface to the underlying operating system, window management system, and network. Attempting to export an instance of these objects results in an exception.

BOSS provides a protocol that is similar to Stream. A root object and the objects it references are stored as a single unit that is maintained by the BOSS archive. Root objects are appended to an archive stream and are retrieved from the stream in the same order in which they were appended. Each root object appended to the stream is referred to as an *object structure*. It is not necessary to understand the internal format of the multiple object structures that are appended to, and retrieved from, a BOSS archive, but it can be important to understand how BOSS identifies and maintains the individual object instances within the object structure and how instances are referenced between structures within the same archive. These issues will be discussed throughout this article.

Each binary representation of an object is stored and referenced by an index. Each unique object instance is assigned an *object index* within the archive. The object index is similar to the value returned by Object>>asOop except that the scope of the BOSS object index is limited to the scope of the BOSS archive and is unique within the archive.

An instance of BinaryObjectStorage that maintains two

Business Graphics, New for Smalltalk/V®

WidgetKit™/Business Graphics brings industry-leading business graphics to Smalltalk/V®. The 11 major types of charts and graphs are as easy to use as WindowBuilder™ Pro/V. WidgetKit/Business Graphics (WK/BG) has high-performance DLLs*, Smalltalk wrappers that integrate the controls into Digitalk's SubPane hierarchy, and Smalltalk classes that allow you to build the UIs using the controls interactively in WindowBuilder Pro/V. With WK/BG you'll quickly build powerful charts and graphs that communicate information the way your users want to see it.

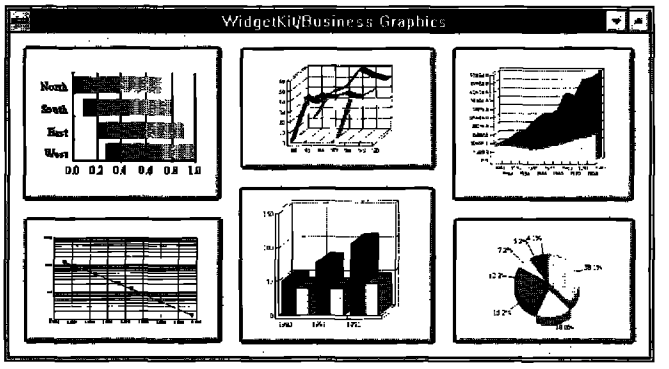


Chart Types in WK/BG (partial)

- **Bar** *2-D & 3-D, horizontal, vertical, simple, stacked*
- **Pie** *2-D, 3-D, regular, exploded*
- **Area** *2-D, 3-D, absolute or %*
- **Line**
- **Gantt**
- **High-low-close**
- **Tape**
- **Log-linear**
- **Bubble**
- **Scatter**
- **Polar**

Printing, Fonts, Colors, and More

WK/BG provides printing for all charts via a programmatic interface. Appropriate charts have autoscaling if desired. You control fonts, colors, titles, labels, legends, justification, fill patterns, line styles, scaling, identifying symbols, grids, ticks, scroll bars, and more at development or run time.

No runtime fees are required for applications developed with WK/BG. Includes complete documentation, full source, and free support to registered users for the first 90 days.

WIDGETKIT/BUSINESS GRAPHICS

NEW!	**For Win**	**$495**
NEW!	**For Win32**	**$495**
	For OS/2	*(1Q95)*

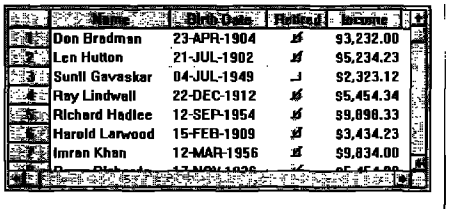
WidgetKit/Business Graphics requires WindowBuilder Pro/V. *Underlying functionality for Win and Win32 is Pinnacle Graphics Server, for OS/2 is Presentation Graphics SDK. WK/BG is compatible with Team/V™ and ENVY®/Developer. Support subscription available.

...And Spreadsheets Too!

WidgetKit™/Professional brings proven and powerful spreadsheet DLLs to Smalltalk/V. And the spreadsheet power is as easy to use as WindowBuilder™ Pro/V. WidgetKit/Professional (WKPro) consists of the FarPoint Professional DLLs, Smalltalk wrappers that integrate the controls into Digitalk's Subpane hierarchy, and Smalltalk classes that allow the controls to be placed and edited interactively in WindowBuilder Pro/V. WKPro enables you to quickly build solid, powerful, reusable, and maintainable UIs for your Smalltalk/V applications.

Graphical Widgets

WKPro includes graphical controls to display pictures (BMP, PCX, & GIF) in spreadsheet cells or separately. Animation too.



	Name	Birth Date	Batting	Income
1	Don Bradman	23-APR-1904	☒	$3,232.00
2	Len Hutton	21-JUL-1902	☒	$5,234.23
3	Sunil Gavaskar	04-JUL-1949	☐	$2,323.12
4	Ray Lindwall	22-DEC-1912	☒	$5,454.34
5	Richard Hadlee	12-SEP-1954	☒	$9,898.33
6	Harold Larwood	15-FEB-1909	☒	$3,434.23
7	Imran Khan	12-MAR-1956	☒	$9,834.00

High-Powered Spreadsheets

You get a spreadsheet similar to Microsoft's Excel™: formulas, drag and drop, and row and column resizing. There are 11 cell types, control of color, formatting, multiple selection, and locking. The spreadsheets have printing, load, and save capability. The functionality is factored into a hierarchy of 7 classes. Choose the one that's right for your app.

Virtual Spreadsheets Too

WKPro includes virtual spreadsheet capability that enables you to load only the visible data.

File System Widgets and More

WKPro also includes DirectoryList, DriveList, FileList, and DirectoryFileList controls. You get input validation widgets for the cell types. Use them for spreadsheet cells or by themselves. UIs built with WKPro are portable to all the supported platforms.

No Runtime Fees

No runtime fees for applications developed with WKPro. It includes complete documentation, full source, and free support to registered users for the first 90 days.

WIDGETKIT/PROFESSIONAL

NEW!	**For Win**	**$395**
NEW!	**For Win32**	**$395**
	For OS/2	**$495** *(1Q95)*

WidgetKit/Professional requires WindowBuilder Pro/V. All the DLL functionality of FarPoint Professional is packaged for easy use in WinbowBuilder Pro/V. WKPro is compatible with Team/V™ and ENVY®/Developer. Support subscription available.

SHARE OBJECT SYSTEMS INC.

Objectshare Systems, Inc.
5 Town & Country Village, Suite 735
San Jose, CA 95128-2026
Fax 408-970-7282
CompuServe 76436,1063

Call to order today (408) 970-7280

or call for free information, 9 AM to 5 PM PST, M-F
30-day money-back guarantee

Persistent Object Management

IdentityDictionaries of associations between objects and indexes: the writerMap and the readerMap. The writerMap maintains the association of object -> index. This table is used while the archive is being constructed in determining if a newly referenced object has already been written into the archive. If it has, the object's object index is written in it's place and the trace of that object is ended. On the other hand, if the object is not present in the writerMap, an index is created and added to the IdentityDictionary. Then the newly referenced object is traced and its contents added to the archive. The readerMap maintains the association of index -> object in the archive. When an existing archive is opened, the readerMap is built up from its contents.

Another useful detail of BOSS internals is the manner in which objects, classes, and class versions are represented. Every object stored in the archive is recorded with its size, class, and identity hash value, along with the binary data and references that define its current state. An object's class is also recorded

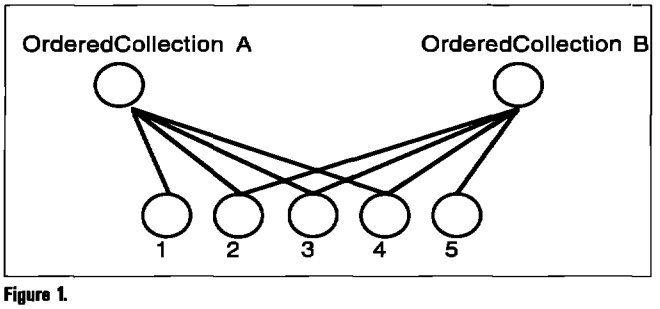


Figure 1.

with the object the first time the class is encountered when building the archive. Information recorded about the class include its *class version number*. When the object is restored, the version number of an archived object's class is compared to the version number of the class present in the image. If the class version number recorded in the archive for the object does not match the class currently present in the image, BOSS cannot restore the object, and a signal is raised. Mechanisms have been provided in BOSS to deal with this situation and are described in the section *Class Versioning*.

In review, each object instance is stored only once in the archive and all references to any object are implemented through the object's unique BOSS object index. In the collections shown in Figure 1, collections A and B store instances of the objects drawn below them. Both collection instances share object instances 2, 3, and 4. When collection A is stored in an archive, BOSS stores both the collection and its object instances in a object structure appended to the achieve stream. Within the object structure, collection A and instances 1, 2, 3, and 4 are assigned indexes and are stored in BOSS binary representation. A side effect of this operation is that the writerMap maintained by the BinaryObjectStorage instance is updated with these five objects and their indexes. When Collection B is stored, it is also appended as an object structure. The collection is traced as a root object, and itself and the objects it maintains are examined and stored as instances in the structure *only if they are not currently present.* Because instances 2, 3, and 4 are already present in the archive, only instance 5 and Collection B are stored as instances and assigned unique indexes. When the structure of Collection B is restored from the archive, it will share the instances of objects 2, 3, and 4 with Collection A—as it should.

This technique of referencing objects through unique indexes within the archive is instrumental in ensuring that the original structure of the set of stored objects is preserved when the archive is later restored. Indexing also eliminates problems with respect to cycles in the references between objects. If while examining the structure of an object being traced, BOSS determines that a referenced object is already present in the archive, it records its presence in the structure using its index and does not further examine this already present object.

There are some caveats and limitations of BOSS's ability to store and retrieve the original structure of the objects stored in an archive. These are described in later sections of this article. But first an example is presented to demonstrate how BOSS is applied by the user.

APPLYING BOSS

BOSS implements a protocol that is a subset of Stream. BOSS operates by appending and retrieving object structures, and the root object they maintain, from a file stream instance:

```
aStream := (Filename named: aFileName) writeStream.
```

An archive is created for a given WriteStream as in:

```
aBOSS := BinaryObjectStorage onNew: aStream
```

The onNew: method initializes the stream by writing an identifying header and BOSS version number at the head of the stream.

```
aBOSS := BinaryObjectStorage onOld: aStream.
```

The onOld: method allows the user to read or append to the stream. BinaryObjectStorage verifies that the stream is a BOSS

archive through the stream's header and BOSS version. BOSS then scans the entire stream to determine the next object index to be assigned in the archive. The stream is then reset to a position *just after* the BOSS header. If you wish to append data to the stream, you must set the file pointer to the end of the stream with the following:

```
aBOSS setToEnd.
```

As mentioned earlier, the stream maintained by BOSS contains a series of object structures. Each object structure is appended to the stream with the following:

```
aBOSS nextPut: anObject.
```

The stream position operations performed by BOSS are performed on the object structures it maintains. Operations like position:, reset, and setToEnd place the stream's position reference at the beginning of an object structure within the stream's contents. For example, the object structure at the stream's current position reference can be retrieved using the following:

```
anObject := aBOSS next.
```

Naturally, once this operation completes, the stream's position reference is located at the next object structure appended to the stream.

It is the responsibility of the user to ensure that each root object retrieved is restored into the correct context within the image being manipulated. For example, if the developer is archiving three global variables as individual root objects in the archive, it is necessary that the application retrieve and reassign these objects in the same order in which they are appended to the archive.

One exception to this explicit assignment of a restored root object's value is that of class objects appended to the archive with BinaryObjectStorage>>nextPutClasses: and retrieved with BOSS>>nextClasses. Class objects archived and restored with these methods are automatically registered with the system dictionary.

Collections of objects to be appended to the archive can be managed in one of two ways. A collection that is appended to a stream as an object is retrieved as a collection with all its members intact. But if the user wishes to archive the contents of a collection as a series of object structures, then the following method is applied:

```
aBOSS nextPutAll: aCollection.
```

In the above code fragment, each member of the collection is appended as a separate object structure as in the following:

```
aCollection do: [:item| aBOSS nextPut: item].
```

A common practice is for an application to organize a set of the set of objects that are to be persistently stored in a file as a collection. This collection of objects is archived using the BOSS>>nextPut: operation. The entire collection can then be retrieved in a single BOSS>>next operation. This process is demonstrated in the following example.

EXAMPLE APPLICATION

In this example, we have an application that maintains a collection of Employee instances as defined by the following:

```
Object subclass: #Employee
    instanceVariables: ' name ssn department '
    classVariables: ' '
    ...
```

Suppose that this application maintains several dictionaries of Employee, each collection representing the employees for a single company. And that each company dictionary is keyed on the social security number of the employee.

If we manage each company dictionary as a global variable (e.g., AcmeWidget and JacksHydraulics), each collection instance can be separately archived and retrieved with the following:

```
(BinaryObjectStorage
    onNew: (Filename named: 'acmewidget.bos') writeStream)
        nextPut: AcmeWidget;
        close.
```

If this operation were repeated for each global variable maintaining a company employee dictionary, then we would have *N* separate files, each containing the Employees of a single company. When each dictionary is restored to its global variable, each of its employees is also restored.

```
aBOSS := BinaryObjectStorage
    onOld: (Filename named: 'acmewidget.bos') writeStream).
AcmeWidget := aBOSS next.
aBOSS close.
```

But a problem arises if it is possible for two or more companies to share a single instance of an employee. In this case, the previous example destroys these semantics when each company is retrieved from its separate archive. This is because BOSS creates a unique instance of the common employee in each company employee dictionary.

This issue is illustrated in Figure 2. In the upper portion, each of the two employee dictionaries shares the same instance of the employee, as indicated by its unique SSN across all

Persistent Object Management

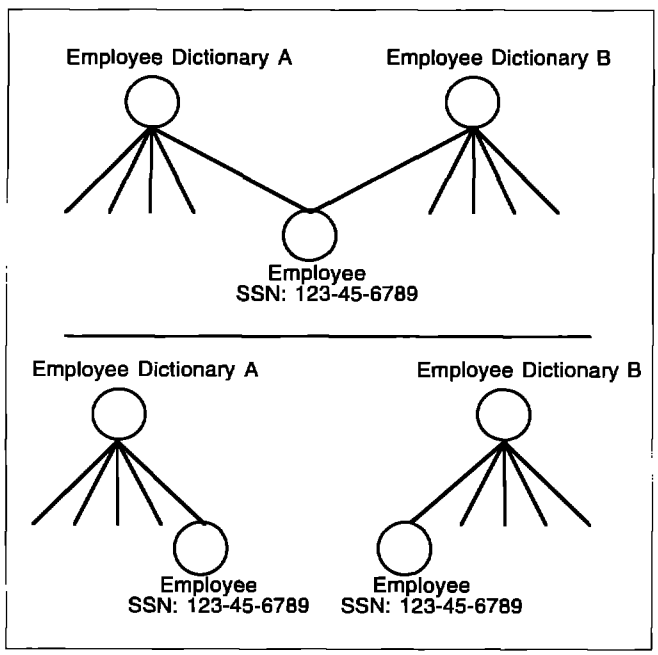


Figure 2.

instances of Employees: But once each dictionary has been stored and retrieved for separate BOSS archives, unique instances of Employee are present in each dictionary.

This example demonstrates the importance of archiving objects in a manner that allows BOSS to note and maintain the true relationships between the object instances being stored. If instead one wished to maintain the membership of an employee in two or more companies, then each company would need to be stored in the same archive, as in the following:

```
aStream := (Filename named: aFileName) writeStream.
aBOS := BinaryObjectStorage onNew: aStream.
aBOS nextPut: AcmeWidget.
aBOS nextPut: JacksHydraulics.
aBOS close.
```

Or perhaps a better implementation would maintain all instances of company employee dictionaries in a single global dictionary Companies, keyed on each company's name. In this case, all dictionaries could be archived and restored with the following:

```
(BinaryObjectStorage
    onNew: (Filename named: companydict.bos') writeStream)
        nextPut: CompanyEmployees;
        close.
aBOSS := BinaryObjectStorage
    onOld: (Filename named: companydict.bos') writeStream).
CompanyEmployees:= aBOSS next.
aBOSS close.
```

In either implementation, an Employee instance maintained by two or more company dictionaries will share a common object index in the BOSS archive and will be restored as a single Employee instance present in all interested dictionaries.

These examples demonstrate how BOSS can be applied to

storing and restoring a collection of arbitrary object instances and the relationships they share. The following section describes how to deal with retrieving objects whose class definition has been modified.

CLASS VERSIONING

A problem arises when an object is to be retrieved from an archive whose class has been modified and is no longer compatible with the format of the archived object. For example, suppose in our example Employee class, we added an instance variable 'age', as in the following:

```
Object subclass: #Employee
    instanceVariables: ' name ssn age department '
    classVariables: ' '
    ...
```

Now Employee instances that are currently stored in a BOSS archive are incompatible with the current class definition. BOSS detects this incompatibility when the object is retrieved from the archive by comparing the class format stored with the object in the BOSS object structure with the current format of the class as defined in the image.

BOSS provides a mechanism that allows the class itself to handle this situation. If the class defines the class method binaryReaderBlockForVersion:format:, this method will be called when the BOSS detects a difference between the format of the class of the object in the archive and the format of the class present in the image. This method is expected to return a block that will convert an instance of the old version into the new version.

For example:

```
Employee class methodsFor: BOSS
binaryReaderBlockForVersion: ver format: fmt
    | emp |
    (ver = 'Employee 3/93') ifTrue: [
        ^[:oldVers |
            emp := Employee new
            ssn: (oldVers at: 1) ;
            department: (oldVers at: 2) ;
            age: 99. "method of obtaining age needed!"
            oldVers become: emp]].

    ^ super binaryReaderBlockForVersion: ver format: fmt.
```

This code fragment demonstrates several interesting facets of this BOSS facility. First, note that the version argument ver is used to determine the version of the object's class stored in the archive. This version information is provided by the class through the following class method:

```
binaryRepresentationVersion
    ^'Employee 6/94'.
```

In our example, the older class version is identified by the string 'Employee 3/93' but could be any immediate value.

The information maintained about an object's class in the BOSS archive includes the value returned by this class method if the method is present in the class definition. Our example is only checking for a single class version. But if the class has been updated two or more times, each different class version could be handled by a different block returned from binaryReaderBlockForVersion:format:. The format: argument is normally not used and is not described here.

The block returned from binaryRea...ion:format: should expect a single argument. For pointer-type objects (like Employee), the argument will contain an array of instance variables from the archived object. For byte-type objects, the argument passed to the block will be a byte string of the archived object's contents. In either case, the block is responsible for:

1. Creating an instance of the new class
2. Assigning instance variables from the array to the correct instance variables of the new object or copying the contents of the byte string into the new object
3. Assigning default values for instance variables not present in the old format, and, finally
4. The object referenced by the block argument (array or byte string) is converted into the new object using the become: method.

> ***"BOSS was developed to provide a service whereby large numbers of objects can be persistently stored and/or moved between images. It is not intended to replace a database or general purpose persistent object management system."***

In review, BOSS provides a mechanism for detecting versioning differences between the class of objects maintained in an archive and the class currently maintained in the image. This mechanism is implemented in the class method binaryReaderBlockForVersion:Format:. This method is expected to determine the class version of the object maintained by the archive and returns a block that will be applied to converting the old version object into the new version. The value returned from the class method binaryRepresentationVersion is used to identify the class version of every object stored in the archive. Any class that expects to utilize BOSS should define this method in anticipation of the class being modified.

BOSS PERFORMANCE & OPTIMIZATION

When creating or retrieving an archive, BOSS can exhibit poor performance when *many thousands of objects* are being maintained in the archive. The reason for this poor performance is the maintenance of the writer and reader maps: the identity dictionaries holding the associations between index and object in the archive. Normally, every object that is stored in the

archive is assigned an index that is maintained by these maps. Each time an object is added to the archive, the writer map is checked to see if the object is already present in the archive (see the section Binary Object Streaming Service). Although this check is quite fast, when the archive is maintaining a hundred thousand or more objects, the archival process can be perceived by the user as being slow.

For example, an application where we employed BOSS to save several large object structures involved storing more than 330,000 objects. The time to save these objects into an archive was more than 22 minutes on an HP715/33 workstation. The archive file size was approximately 1.8 MBytes. The time to restore this archive from file was 9 minutes. In another situation we experienced a 7-minute delay when saving a structure with more than 157,000 objects and required 3 minutes to restore the archive. This latter examples was executed on a HP720/60 workstation. Even though these archival operations are intended to be applied infrequently, we wanted to see how we might improve on this time. Note that an archive involving less that a 10-K objects can be archived in less than 1 minute.

Fortunately, BOSS allows the user to make a tradeoff between archival time and storage space requirements. Because of the bottleneck in manipulating an archive-exist search for objects in the writer and reader maps, BOSS allows the user to specify ranges of indexes to be forgotten. This reduces the search space when adding a new object to the archive.

This mechanism is implemented in the following methods:

```
BOSS >> forgetInterval: anInterval.
BOSS >> forgetInterval: anInterval excluding: anIndex.
```

The first version of this method allows the user to ignore the given interval of object indexes. The second version allows the user to specify an interval of object indexes to forget excluding the root object referenced by anIndex.

The index of an object can be obtained using BOSS >> indexOf: anObject. Alternatively, the next index to be assigned to an object can be retrieved using BOSS >> nextIndex. The following code fragment allows the user to forget the interval of a root object and all its child objects affer storage:

```
start := aBOSS nextIndex.
aBOS nextPut: anObject.
aBOS forgetInterval: (start to: (aBOS nextIndex - 1)).
```

There are two potential problems with this mechanism. Both arise because an object in a forgotten interval will be rewritten into the archive as a new instance if it is re-encountered during the archival process. Storing the object multiple times increases the size of the archive. It also causes two references to the same object to be split into two instances when the index of the first reference has been forgotten. This situation is similar to our Employee example in the previous section. It is important to find a partitioning of objects to be archived so that references to the same object will either not be split or, if the split occurs, it will not matter to the semantics of the application.

When this process was employed in our larger archive (330-K object archive) we were able to find three partitions of 100-K objects, roughly dividing the archive by thirds. We were able to reduce our archival time from 22 to 15 minutes or to 5 minutes for each 100-K object partition. In terms of the space tradeoff, size of the archive increased by less than 100-K bytes. In the case of our smaller archive we were able to find a partitioning of the 150-K objects into 100 archival units. With this use of the forgetInterval: feature we were able to reduce the archival time from 9 to under 1 minute.

CLOSING CONSIDERATIONS

ParcPlace Smalltalk's Binary Object Steaming Services allow the user to persistently store arbitrary structures of objects to a file stream. In an operation similar to a deep copy, the user is able to store and retrieves the structure of a root object plus any objects reachable from the root. Archived object structures are retrieved from the archive using the stream-like protocol provided by the class BinaryObjectStorage.

BOSS was developed to provide a service whereby large numbers of objects can be persistently stored and/or moved between images. It is intended to address the shortcomings of the storeOn: / readFrom: operations. That is, the representation applied by BOSS is more compact that applied by storeOn:, and the structure of the objects reachable from the root object can be an arbitrary graph including cycles.

BOSS is not intended to replace a database or general purpose persistent object management system. It lacks transactions and concurrency control, and objects cannot be accessed using a key or index. An object retrieved from a BOSS archive must be explicitly assigned to the correct context within the application.

When applying BOSS it is important to find the correct partitioning of root objects. Too large a root forces the entire structure to be retrieved from secondary storage when only a small portion of the overall structure is needed. BOSS is most efficient when storing large sets (1,000–30,000) of objects in each root structure as opposed to storing and retrieving many small, single-object instances. However, because BOSS maintains an internal index for every object it manages, an archiving operation can get slow when the archive gets very large (> 75,000 to 100,000+ objects). To address this problem, the service provides mechanisms for restricting the size of the index table that will speed up operations on large archives. ♀

References

1. ObjectWorks Smalltalk Users Guide, Chapter 27, Binary object streaming service, ParcPlace System, Inc.
2. Vegdahl, S.R. Moving structures between Smalltalk images, Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, Portland OR, 1986, pp. 466–471.

Michael Christiansen works at Bell Northern Research, Richardson, TX, where he is developing cellular telecommunications network performance analysis, configuration, and management systems. He is currently developing Smalltalk implementations of distributed network management services, as defined by CMISE and OSI standards. He can be reached by email at mikec@metronet.com.

Object transfer between Smalltalk VMs

Tony White, Dwight Deugo, and Joe Ulvr

UNIX sockets enable UNIX processes to transfer data between one another, and are often used to implement client–server relationships, where a client process sends requests to a server's address at one end of a socket and the server waits for them to arrive at the other end.

There are different types of sockets depending on the UNIX version being used. The two main types of sockets are stream sockets and datagram sockets. Stream sockets provide continuous two-way, byte-stream communication, and datagram sockets transfer fixed, maximum length packets of information. There are four steps to sending and receiving information across a socket:

1. create a new socket,
2. open a connection across it,
3. open a communication stream on the connection, and
4. transmit the information.

In ObjectWorks\Smalltalk, a socket can be used to transfer data from a Smalltalk process to other UNIX processes, including another Smalltalk process on a different virtual machine (VM). This facility is implemented by six standard ObjectWorks\Smalltalk classes:

- UnixSocketAccessor
- IPSocketAddress
- SocketAddress
- UDSocketAddress
- ExternalConnection
- ExternalReadAppendStream.

To find out how to use a socket in Objectworks\Smalltalk one should begin by browsing the class UnixSocketAccessor.[1] This class provides methods for creating sockets and several example methods describing how to use them.

When the sending and receiving processes do not share a common representation, it is convenient for them to transfer data in a standard format, such as a byte-stream or fixed length packets. This format can then be decoded by the receiving process to one that is more suitable. However, when the sending and receiving processes are Smalltalk processes, this approach is inconvenient. There is only one "format" in Smalltalk: an object. Therefore, Smalltalk processes should be able to send and receive any arbitrarily complex object over a socket without being forced to format or decode it from a byte-stream or a sequence of fixed-sized packets.

In this article we discuss how the Binary Object Streaming Service (BOSS)[1] and sockets[2] can be used to provide such a mechanism. The second section describes how to incorporate BOSS with a socket in order to transmit arbitrarily complex objects. The third section describes a client-server architecture that uses our approach for transmitting complex objects. The fourth section contains a comparison of the BOSS-based approach with a string-based approach. The fifth section concludes with a discussion of our approach's benefits and limitations.

AN INTERFACE FOR THE TRANSMISSION OF COMPLEX OBJECTS

A string-based approach to the transmission of complex objects between Smalltalk VMs is to use the storeOn: method to create a string representation of the object, transmit that to the receiving process over a socket, where the evaluate: message is sent to the Compiler class with the received string as the argument. This approach has the dual disadvantages of spending significant amounts of time transforming and compiling the original object to and from a string object.

We describe another approach, implemented in the TransportInterface class, that provides a transport mechanism avoiding these drawbacks through the use of BOSS streams, rather than raw character streams. The TransportInterface class supports asynchronous, nonblocking I/O and can be used in a client–server framework for communication between Smalltalk VMs.

Applications requiring Smalltalk-to-Smalltalk object exchange can create a server TransportInterface by using the newAtPort: class method. This method creates a "listening" socket that can be used to wait for client connections. A client TransportInterface can be generated by using the class message newAtHost:port:. If the client request is successful, the initialization of the connection between client and server occurs when the setupInterface messages are sent to the TransportInterface objects. Examples of client and server interfaces are shown in Listing 1 in the class methods exampleClient and exampleServer respectively.

The setupInterface message performs several important functions. First, an ExternalConnection is created for the interface and the newly created socket is associated with both input and output directions (sockets being bidirectional communication entities). Second, the stream associated with the interface is initialized as being an ExternalReadAppend stream; again reflecting the bidirectional nature of socket I/O. Third, a SharedQueue is created. This is used by the application for writing on the socket. Finally, read and write processes are created for the TransportInterface object; the readProcessLoop and writeProcessLoop methods providing the process functionality. BOSS streams are required for both reading and writing. They are initialized during the first read or first write on the socket. Objects transferred via the TransportInterface class are then

Object Transfer

Table 1.

Approach	Median Image Experiment	Median ByteString Experiment
BOSS-based	9 ms	9 ms
String-based	3636 ms	430 ms

written to and read from the socket using the BOSS nextPut: and next methods respectively. Using these methods allows objects to be written to and read from the socket directly; without any user-provided encoding.

Once setup, asynchronous input is handled by the readProcessLoop code that is now executing inside of a process generated for the interface. Referring now to the readObject method, the process remains blocked (by the sending of the message readWait) in the next method of BOSSReader until input is available on the socket. When input is available, the readProcessLoop process is signaled, the physical read completes, and the object is returned by the BOSSReader next method. If the object received is nil, it implies that the remote end of the connection has closed its end of the socket interface, and, in this situation, the local end closes its socket interface. When non-nil, dependents of the TransportInterface are advised of the arrival of the object using the changed:with: message—the aspect being #socketIO. Hence, it is possible to broadcast the received object to multiple application-level objects by having them be dependents of the TransportInterface object.

Applications sending objects over the interface use the TransportInterface nextPut: message. This method does not write on the socket, but on the shared queue. As shared queue writes are nonblocking, the write—from the point of view of the application process—will also be nonblocking. The physical write of the object to the socket is performed by the writeProcessLoop code that is also executing inside of a process generated for the interface. Referring now to the nextPutDirectly: message sent in the writeProcessLoop method, the writeProcessLoop process is blocked until something is written to the shared queue. Once an object has been written to the shared queue, the writeProcessLoop process is signalled, the next method completes and the nextPutDirectly: message is sent to the TransportInterface. This method performs the physical write to the socket that may also block, possibly because the socket is full. However, once output can be written to the socket, the write process will be signaled, the object is written in BOSS format to the socket and the write process returns to waiting for another object to write at the shared queue.

Applications receive objects sent over the TransportInterface by intercepting update:with:from: messages and performing special processing when the aspect is #socketIO.

Sending the close message to the TransportInterface object causes the read and write processes to be terminated and the socket to be closed. When the socket closes, dependents of the TransportInterface are sent a changed: message with the #closedSocket as aspect.

This architecture—with processes for reading and writing of objects to the socket—provides for asynchronous communication and nonblocking I/O as seen by the application object(s) using the TransportInterface.

CLIENT–SERVER ARCHITECTURE

While the previous section has indicated how a basic connection can be made between two UNIX processes, and how BOSS can be used to simplify the exchange of complex objects between Smalltalk VMs, communication often occurs within a client-server architecture. The SmalltalkInterface, SmalltalkClient and SmalltalkServer classes shown in Listings 2–4, respectively, along with the previously described TransportInterface class, provides a set of classes that can be used in the construction of client-server applications.

The SmalltalkInterface class is an abstract class. It provides support for common client-server behavior. For example, the close method closes the TransportInterface, the update:with:from: method provides special processing of the #socketIO aspect of changes, and the nextPut:on: method sends an object on a specific TransportInterface. Two instance variables—application and callback—are used to store the application object and method to which the received object is to be dispatched.

An instance of a SmalltalkClient is generated by sending the newAtHost:port: message to the class SmalltalkClient. If a connection to a server on the requested host and at the appropriate port can be established, a correctly initialized SmalltalkClient object is returned. Once successfully created, objects can be sent to the server by sending the nextPut: message to the SmalltalkClient object. When communication is no longer required, sending the close message to the SmalltalkClient object closes the underlying TransportInterface.

An instance of a SmalltalkServer is more complicated than a corresponding SmalltalkClient instance because of the requirement to listen for new client-server connections. A listener process is created to perform this task. The listener process code—found in the method listenerProcessLoop:—remains blocked within the code executed by the accept message until a new client requests a connection. At this time the listener process is signaled and a new TransportInterface is created and added to the list of clients supported by the SmalltalkServer. What might appear strange about the server socket code segment in the method listenerProcessLoop: is that the original socket is not used for the two-way communication. Its sole purpose is to accept connections at a specific port. Once a connection is established by the socket, it returns another socket, called childSocket in the aforementioned code, to use for two-way communication.

RESULTS

To demonstrate the efficiency of the BOSS-based approach, we ran two experiments. The first one transmitted a 500×500 image, with all bits set, from a client to a server, which responded with a ByteString of 20 characters. The second experiment transmitted a 10,000 random character ByteString from a client to a server, which responded with a ByteString of 20 characters. Each experiment ran 101 times, and the total time for a client to send and receive the objects was recorded.

We ran the first experiment using our BOSS-based approach contrasting it with the string-based approach described at the beginning of the second section. However, since the second

experiment involved transmitting only strings, the storeOn: at a client end and compiler evaluation at the server end of the string-based approach were removed. Table 1 describes the median timing results of these experiments.

Table 1 shows that for large objects, like an image, our approach runs 404 times faster, and, when sending only strings, it still ran 47 times faster than the string-based approach.

One of the major gains in efficiency is a result of the BOSS-based approach not having to compile a string to regenerate the object. On the server end, it becomes a very expensive operation for the compiler to parse a string representing a very large object, and on the client end it is also an expensive operation to generate the definition for an object in the form of a string. Even if these tasks are removed and we only transmit strings as in the second experiment, the BOSS-based approach is still faster!

CONCLUSIONS

There are other approaches one can use to transmit complex objects between Smalltalk VMs.

For example, one approach is to use HP's Distributed Smalltalk,[3] where the ability to pass objects between VMs is built into the architecture. Another is to use ENVY/Swapper,[4] which is a high speed object loader/unloader that provides a method of storing and retrieving objects between all supported virtual machines and platforms. However, if one does not have these software packages available, our approach provides an elegant, inexpensive and efficient method of object transmission.

Our approach is not without its limitations. One is that the class definitions of the transmitted objects must exist at the receiving end of the socket. Another is that class and metaclass objects can not be transmitted. Both of these limitations are a result of a chicken-and-egg situation. One cannot make use of a class until it exists, and, even if you attempt to transmit the class information, the class has to be defined before one can refer to it. Naturally, objects that do not have a BOSS representation cannot be transmitted using this approach.

We have described a simple, easy-to-use, interface between Smalltalk VMs that can be used to send objects over sockets without user-provided encoding. The interface provides asynchronous, nonblocking I/O in a client-server framework, and, through simple experimentation, it has been shown to be considerably more efficient when compared to string-based encodings. ♀

References

1. ParcPlace Systems, Inc. ObjectWorks\Smalltalk User's Guide, ch. 7, 1992.
2. ParcPlace Systems, Inc. ObjectWorks\Smalltalk User's Guide, ch. 23, 1992: 234.
3. Hewlett-Packard Company, HP Distributed Smalltalk Reference Guide, 1993.
4. Object Technology International, Inc. ENVY/Swapper High-Speed Object Loader/Unloader Manual, Release 3.50, 1993.

Tony White is a member of the Computing Research Lab at Bell-Northern Research. He can be reached at Bell-Northern Research in Ottawa, ON, Canada, at 613.765.4279, or by email at arpw@bnr.ca.

Dwight Deugo is a consultant with The Object People. He can be reached at The Object People, Ottawa, ON, Canada, at 613.225.8812, or by email at dwight@ObjectPeople.on.ca.

Joe Ulvr is a third-year electrical engineering student at the University of Waterloo, Waterloo, ON, Canada. He can be reached by email at jlulvr@electrical.watstar.uwaterloo.ca.

Listing 1.

```
class: TransportInterface
superclass: Model
instance variables: `socket connection stream bossWrite bossRead
readProcess writeProcess writeQueue ` class variable: `'
poolDictionaries: `'
category: `Smalltalk-Interface'

TransportInterface instance methods

`initialize-release'

close
    self closeRead; closeWrite; closeStream; closeSocket.

closeRead
    | aProcess |
    readProcess notNil
        ifTrue:
            [aProcess := readProcess.
            aProcess == Processor activeProcess
                ifTrue: [[self close] fork]
                ifFalse: [
                    readProcess := nil
                    aProcess terminate]]

closeWrite
    | aProcess |
    writeProcess notNil
        ifTrue:
            [aProcess := writeProcess.
            aProcess == Processor activeProcess
                ifTrue: [[self close] fork]
                ifFalse: [
                    writeProcess := nil
                    aProcess terminate]]

closeStream
    | aStream |
    stream notNil
        ifTrue:
            [aStream := stream.
            stream := nil.
            self changed: #closedSocket.
            aStream close]

closeSocket
    | aSocket|
    socket notNil
        ifTrue:
            [aSocket := socket.
            socket := nil.
            aSocket close]

`message processing'

readProcessLoop
    | delay |
    delay := Delay forMilliseconds: self class timeout.
    [socket successful] whileTrue: [
        self processObject.
        delay wait]

writeProcessLoop
    | delay |
    delay := Delay forMilliseconds: self class timeout.
    [socket successful] whileTrue: [
        self nextPutDirectly: self writeQueue next.
        delay wait]

`private'

processObject
    | anObject |
    anObject := self readObject.
    anObject isNil
        ifTrue: [self close]
        ifFalse: [self changed: #socketIO with: anObject]

`specialized IO'

nextPut: anObject
    self writeQueue nextPut: anObject.

nextPutDirectly: anObject
    (stream isKindOf: Stream)
        ifTrue: [UnixSocketAccessor errorReporter peerFaultSignal
            handle:
                [:exception |
                self close.
                exception return]
            do:
                [self bossWrite isNil ifTrue: [self bossWriteInit: stream].
                self bossWrite nextPut: anObject.
                stream commit]]

readObject
    stream atEnd ifFalse: [UnixSocketAccessor errorReporter
peerFaultSignal
        handle: [:exception | exception return]
        do:
            [self bossRead isNil ifTrue: [self bossReadInit: self stream].
            ^self bossRead next]].
    ^nil

`private initialization'

bossReadInit: aStream
    self bossRead: (BinaryObjectStorage onOldNoScan: aStream).

bossWriteInit: aStream
    self bossWrite: (BinaryObjectStorage onNew: aStream).

setupInterface
    self setupInterfaceOn: self socket

setupInterfaceOn: aSocket.
    self socket: aSocket.
    self connection: ExternalConnection new.
    self connection input: aSocket; output: aSocket.
    self stream: self connection readAppendStream.
    self readProcess: ([self readProcessLoop] forkAt: Processor activePriority - 1).
    self writeProcess: ([self writeProcessLoop] forkAt: Processor activePriority - 1).
    self writeQueue: SharedQueue new.

TransportInterface class methods

`instance creation'

newAtHost: aHost port: aPortNumber
    UnixSocketAccessor errorReporter peerFaultSignal
        handle:
            [:exception |
                Dialog warn: `Server at port number ` ,
                    aPortNumber printString , ` not running on host `,
                    aHost, `.'.^nil]
        do:
            [^self newOn: (UnixSocketAccessor newTCPclientToHost:
                aHost port: aPortNumber)]

newAtPort: aPortNumber
    | aSocket |
    OSErrorHolder errorReporter invalidArgumentsSignal
        handle:
            [:exception |
            Dialog warn: `Server at port number ` ,
```

```
                aPortNumber printString , ` cannot be created.'.
            ^nil]
        do:
            [aSocket := UnixSocketAccessor newTCPserverAtPort: aPortNumber.
            aSocket listenFor: self queueSize.
            ^self new socket: aSocket; yourself]

newOn: aSocket
    ^self new setupInterfaceOn: aSocket

`class constants'

timeout
    ^100.

queueSize
    ^1

`examples'

exampleClient
    | aClient |
    aClient := self newAtHost: `localhost' port: 3603.
    aClient notNil ifTrue: [
        aClient
            setupInterface;
            addDependent: SocketMsgReceiver new.
        aClient nextPut: `Hello from the client'.
        (Delay forSeconds: 1) wait.
        aClient close]

exampleServer
    | aClientInterface aServer |
    aServer := self newAtPort: 3603.
    aServer notNil ifTrue: [
        aClientInterface := self newOn: aServer socket accept.
        aClientInterface addDependent: SocketMsgReceiver new.
        aClientInterface nextPut: `Hello from server'.
        (Delay forSeconds: 1) wait.
        aClientInterface close.
        aServer close]
```

Listing 2.

```
class: SmalltalkInterface
superclass: Model
instance variables: `application callback transport `
class variables: `'
poolDictionaries: `'
category: `Smalltalk-Interface'

SmalltalkInterface instance methods

`initialize-release'

close
    self transport close.
    self transport: nil.

`updating'

update: aSymbol with: anObject from: aSender
    aSymbol == #socketIO
        ifTrue: [(self application respondsTo: self callback)
            ifTrue: [self application perform: self callback with: anObject].
            dependents update: aSymbol with: anObject from: aSender]
        ifFalse:
                [super
                update: aSymbol
                with: anObject
                from: aSender]

`specialized IO'

nextPut: anObject on: aTransportInterface
    aTransportInterface nextPut: anObject.

SmalltalkInterface class methods

`examples'

exampleClientAtPort: aPort
    | aClient |
    aClient := SmalltalkClient newAtHost: `localhost' port: aPort.
    aClient notNil ifTrue: [
        aClient
            application: aClient;
            callback: #callbackMethod:;
            nextPut: `Hello from client';
            nextPut: (OrderedCollection with: ColorValue black with: 2.9);
            nextPut: `I am done'.
        (Delay forSeconds: 6) wait.
        aClient close]

exampleServerAtPort: aPort
    | aServer |
    aServer := SmalltalkServer newAtPort: aPort.
    aServer notNil ifTrue: [
        aServer application: aServer; callback: #callbackMethod:]

`cleanup'

release
    self allInstances do: [:aSmalltalkInterface |aSmalltalkInterface close].
    ObjectMemory globalGarbageCollect.
```

Listing 3.

```
class: SmalltalkClient
superclass: SmalltalkInterface
instance variables: `'
class variables: `'
poolDictionaries: `'
category: `Smalltalk-Interface'

SmalltalkClient instance methods

`specialized IO'

nextPut: anObject
    self nextPut: anObject on: self transport.

`updating'

update: aSymbol with: anObject from: aSender
    aSymbol == #closedSocket
    ifTrue: [self close]
        ifFalse: [super update: aSymbol with: anObject from: aSender]

SmalltalkClient class methods

`instance creation'

newAtHost: aHost port: aPort
    | anInterface aTransport |
    aTransport: := TransportInterface newAtHost: aHost port: aPort.
    ^aTransport notNil
        ifTrue: [
            anInterface := self new.
            (anInterface transport: aTransport) setupInterface.
            aTransport addDependent: anInterface]
        ifFalse: [nil].
```

Listing 4.

```
class: SmalltalkServer
superclass: SmalltalkInterface
instance variables: `listenerProcess interfaces ` class variables: `'
poolDictionaries: `'
category: `Smalltalk-Interface'

SmalltalkServer instance methods

`private'

closeInterfaces
    (interfaces respondsTo: #do:) ifTrue: [
```

continued on page 31

KENT BECK

Simple Smalltalk testing

You can't argue with inspiration (or deadlines). I started to write the final column in the sequence about using patterns for design, but what came out was this. It describes some work I have been doing with a framework that takes the tedium out of writing tests. I'll get back to the pattern stuff in the next issue.

Smalltalk has suffered because it lacks a testing culture. This column describes a simple testing strategy and a framework to support it. The testing strategy and framework are not intended to be complete solutions, but, rather, are intended to be starting points from which industrial strength tools and procedures can be constructed.

The article is divided into four sections:

- *Philosophy.* Describes the philosophy of writing and running tests embodied by the framework. Read this section for general background.
- *Framework.* A literate program version of the testing framework. Read this for in-depth knowledge of how the framework operates.
- *Example.* An example of using the testing framework to test part of the methods in Set.
- *Cookbook.* A simple cookbook for writing your own tests.

PHILOSOPHY

The most radical philosophy espoused here is a rejection of user-interface-based tests. In my experience, tests based on user interface scripts are too brittle to be useful. Testers spend more time keeping the tests up to date and tracking down false failures and false successes than they do writing new tests.

My solution is to write the tests (and check results) in Smalltalk. Although this approach has the disadvantage that your testers need to be able to write simple Smalltalk programs, the resulting tests are much more stable.

Failures and errors

The framework distinguishes between failures and errors. A failure is an anticipated problem. When you write tests, you check for expected results. If you get a different answer, that is a failure. An error is more catastrophic; it indicates an error condition that you didn't check for.

Kent Beck has been discovering Smalltalk idioms for eight years at Tektronix, Apple Computer, and MasPar Computer. He is the founder of First Class Software, which develops and distributes reengineering products for Smalltalk. He can be reached at First Class Software, P.O. Box 226, Boulder Creek, CA 95006-0226, or at 408.338.4649 (phone), 408.338.3666 (fax), 70761,1216 (Compuserve).

Unit testing

I recommend that developers write their own unit tests, one per class. The framework supports the writing of suites of tests, which can be attached to a class. I recommend that all classes respond to the message testSuite, returning a suite containing the unit tests. I recommend that developers spend 25–50% of their time developing tests.

Integration testing

I recommend that an independent tester write integration tests. Where should the integration tests go? The recent movement of user-interface frameworks to better facilitate programmatic access provides one answer—drive the user interface, but do it with the tests. In VisualWorks (the dialect used in the implementation below), you can open an ApplicationModel and begin stuffing values into its ValueHolders, causing all sorts of havoc, with very little trouble.

Running tests

One final bit of philosophy. It is tempting to set up a bunch of test data, then run a bunch of tests, then clean up. In my experience, this procedure always causes more problems that it is worth. Tests end up interacting with one another, and a failure in one test can prevent subsequent tests from running. The testing framework makes it easy to set up a common set of test data, but the data will be created and thrown away for each test. The potential performance problems with this approach shouldn't be a big deal, because suites of tests can run unobserved.

FRAMEWORK

The smallest unit of testing is the TestCase. When a TestCase runs, it sets up its test data, runs a test method, then discards the test data. Because many cases may want to share the same test data, TestCase chooses which method to run with the instance variable selector, which will be performed to run the test method.

```
Class: TestCase
    super class: Object
    instance variables: selector
    class variable: FailedCheckSignal
```

TestCases are always created with a selector. The class method selector: ensures this.

```
TestCase class>>selector: aSymbol
    ^self new setSelector: aSymbol
TestCase>>setSelector: aSymbol
    selector := aSymbol
```

The simplest way to run a TestCase is just to send it the message run. run invokes the set up code, performs the selector, then runs the tear-down code. Notice that the tear-down code is run regardless of whether there is an error in performing the test.

```
TestCase>>run
    self setUp.
    [self performTest]
        valueNowOrOnUnwindDo: [self tearDown]!
```

Subclasses of TestCase are expected to create and destroy test fixtures in setUp and tearDown, respectively. TestCase itself provides stubs for these methods that do nothing:

```
TestCase>>setUp
    "Run whatever code you need to get ready for the test to run."
TestCase>>tearDown
    "Release whatever resources you used for the test."
```

PerformTest just performs the selector:

```
TestCase>>performTest
    self perform: selector
```

A single TestCase is hardly ever interesting once you have gotten it running. In production, you will want to run suites of TestCases. Aggregating TestCases is the job of the TestSuite:

```
Class: TestSuite
    super class: Object
    instance variables: name testCases
```

When a TestSuite is created, it is initialized to prepare it to hold TestCases. TestSuites are also named, so you can identify them even if you have, for example, read them in from secondary storage:

```
TestSuite class>>named: aString
    ^self new setName: aString
TestSuite>>setName: aString
    name := aString.
    testCases := OrderedCollection new
```

TestSuites have an accessing method for their name in anticipation of user interfaces that will have to display them:

```
TestSuite>>name
    ^name
```

TestSuites have protocol for adding one or more TestCases:

```
TestSuite>>addTestCase: aTestCase
    testCases add: aTestCase
TestSuite>>addTestCases: aCollection
    aCollection do: [:each | self addTestCase: each]
```

When you run a TestSuite, you'd like all of its TestCases to run. It's not quite that simple, though. Running a suite is different from running a single test case. For example, if you have a suite that represents the acceptance test for your application, after it runs, you'd like to know how long the suite ran and which of the cases had problems. This is information you would like to be able to store away for future reference.

TestResult solves this problem. Running a TestCase just executes the test method and returns the TestCase. Running a TestSuite, however, returns a TestResult that records the information described above- the start and stop times of the run, the name of the suite, and any failures or errors:

```
Class: TestResult
    super class: Object
    instance variables: startTime stopTime testName failures errors
```

When you run a TestSuite, it creates a TestResult, which is time stamped before and after the TestCases are run:

```
TestSuite>>run
    | result |
    result := self defaultTestResult.
    result start.
    self run: result.
    result stop.
    ^result
```

The default TestResult is constructed by the TestSuite:

```
TestSuite>>defaultTestResult
    ^self defaultTestResultClass test: self
TestSuite>>defaultTestResultClass
    ^TestResult
```

A TestResult is always created on a TestSuite:

```
TestResult class>>test: aTest
    ^self new setTest: aTest
TestResult>>setTest: aTest
    testName := aTest name.
    failures := OrderedCollection new.
    errors := OrderedCollection new
```

TestResults are timestamped by sending them the messages start and stop:

```
TestResult>>start
    startTime := Date dateAndTimeNow
TestResult>>stop
    stopTime := Date dateAndTimeNow
```

When a TestSuite runs for a given TestResult, it simply runs each of its TestCases with that TestResult:

```
TestSuite>>run: aTestResult
    testCases do: [:each | each run: aTestResult]
```

Because the selector run: is the same in both TestSuite and TestCase, it is trivial to construct TestSuites which contain other TestSuites, instead of or in addition to containing TestCases.

When a TestCase runs for a given TestResult, it should either silently run correctly, add an error to the TestResult, or add a failure to the TestResult. Catching errors is simple—use the system-supplied errorSignal. Catching failures must be supported by the TestCase itself. First, we need a Signal:

```
TestCase class>>initialize
    FailedCheckSignal := self errorSignal newSignal
        notifierString: 'Check failed - ';
        nameClass: self message: #checkSignal
```

Now we need a way of accessing it:

```
TestCase>>failedCheckSignal
    ^FailedCheckSignal
```

Now, when the TestCase runs with a TestResult, it must catch errors and failures and inform the TestResult, and it must run the tearDown code regardless of whether the test executed correctly. This results in the ugliest method in the framework, because there are two nested error handlers and valueNowOrOnUnwindDo: in one method:

```
TestCase>>run: aTestResult
    self setUp.
    [self errorSignal
        handle: [:ex | aTestResult error: ex errorString in: self]
        do: [self failedCheckSignal
            handle: [:ex | aTestResult failure: ex errorString in: self]
            do: [self performTest]]]
                valueNowOrOnUnwindDo:
                    [self tearDown]
```

When a TestResult is told that an error or failure happened, it records that fact in one of its two collections. For simplicity, the record is just a two element array, but it probably should be a first-class object with a time stamp and more details of the blowup:

```
TestResult>>error: aString in: aTestCase
```

```
        errors add: (Array with: aTestCase with: aString)
    TestResult>>failure: aString in: aTestCase
        failures add: (Array with: aTestCase with: aString)
```

The error case gets invoked if there is ever an uncaught error (for example, message not understood) in the testing method. How do the failures get invoked? TestCase provides two methods that simplify checking for failure. The first, should: aBlock, signals a failure if the evaluation of aBlock returns false. The second, shouldnt: aBlock, does just the opposite.

```
    should: aBlock
        aBlock value ifFalse: [self failedCheckSignal raise]
    shouldnt: aBlock
        aBlock value ifTrue: [self failedCheckSignal raise]
```

Testing methods will likely run some code, then check the results inside should: and shouldnt: blocks.

EXAMPLE

Okay, that's how it works, but how do you use it? Here's a short example that tests a few of the messages supported by Sets. First we subclass TestCase, because we'll always want a couple of interesting Sets around to play with:

```
    Class: SetTestCase
        super class: TestCase
        instance variables: empty full
```

Now we need to initialize these variables, so we subclass setUp.

```
    SetTestCase>>setUp
        empty := Set new.
        full := Set with: #abc with: 5
```

Now we need a testing method. Let's test to see if adding an element to a Set really works:

```
    SetTestCase>>testAdd
        empty add: 5.
        self should: [empty includes: 5]
```

Now we can run a test case by evaluating:

```
    (SetTestCase selector: #testAdd) run.
```

Here's a case that uses shouldnt:. It reads "after removing 5 from full, full should include #abc and it shouldn't include 5."

```
    SetTestCase>>testRemove
        full remove: 5.
        self should: [full includes: #abc].
        self shouldnt: [full includes: 5]
```

Here's one that makes sure an error is signaled if you try to do keyed access:

```
    SetTestCase>>testIllegal
        self should: [self errorSignal
            handle: [:ex | true] do: [empty at: 5. false]]
```

Now we can put together a TestSuite.

```
    | suite |
    suite := TestSuite named: 'Set Tests'.
    suite addTestCase: (SetTestCase selector: #testAdd).
    suite addTestCase: (SetTestCase selector: #testRemove).
    suite addTestCase: (SetTestCase selector: #testIllegal).
    ^suite
```

Figure 1 shows an Object Explorer picture of the suite and of the TestResult we get back when we run it.

The test methods shown above only cover a fraction of the functionality in Set. Writing tests for all the public methods in Set is a daunting task. However, as Hal Hildebrand told me after using an earlier version of this framework, "If the underlying objects don't work, nothing else matters. You have to write the tests to make sure everything is working."

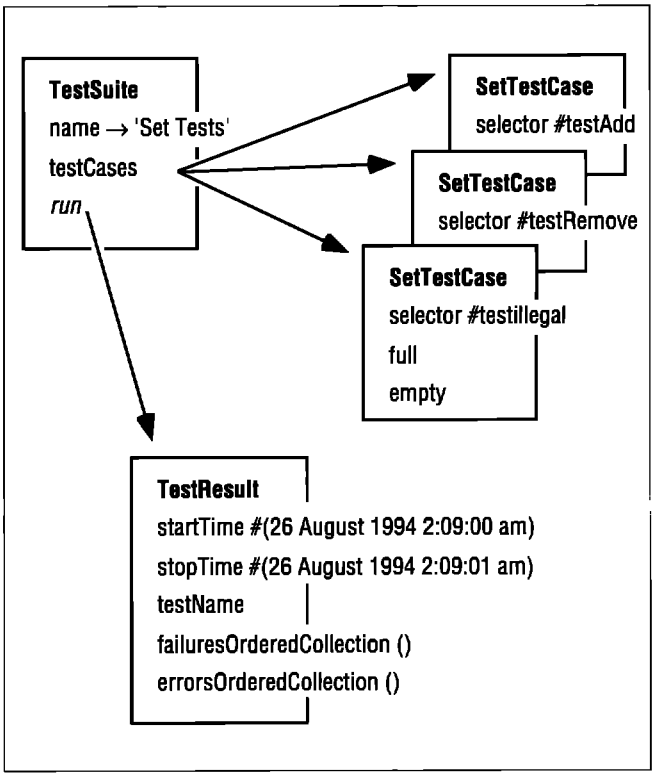


Figure 1.

COOKBOOK

Here are simple steps you can follow to write your own tests:

1. Scope the tests. Decide whether you want to test one object or several cooperating objects. Subclass TestCase, prefixing the name of your test scope.
2. Create test fixture. Find a few configurations of your objects that will exercise the code you want to test. Add an instance variable to your TestCase subclass for each configuration. Override setUp to create the configurations.
3. Write the test methods. Write a method for each distinct test you want to perform. Use should: and shouldnt: wherever you can check for correct or incorrect results. As you write each method, test it by creating an instance of your TestCase subclass and sending it run.
4. Create the suite. Write a method in one of the classes you are testing that collects all of the TestCases into a TestSuite and returns it.

CONCLUSION

This column has presented the philosophy and implementation of a simple testing strategy for Smalltalk. The strategy has the advantages that it is simple, lightweight, and produces tests that are stable. It is not complete or perfect, but it's a whole lot better than no programmatic tests at all. As always, if you have comments, please pass them on to me at 70761.1216@compuserve.com. ♀

ALAN KNIGHT

More performance tips

In retrospect, I wish I'd chosen a different topic for this article. I've just returned from ParcPlace's first user conference, and from listening to Alan Kay's wonderful keynote speech. Alan Kay doesn't talk about performance optimization, except perhaps as one of those petty concerns that prevent us seeing the truly new and exciting ideas. Listening to him talk can make you uncomfortably aware of just how unimportant the problems you're working on are in the larger context. As he so diplomatically put it, "All this crap you're writing code for is meaningless."

This isn't exactly inspirational material for a column on performance optimization, but perhaps there's a lesson here. Performance optimization is a detail. It's easy to get carried away with optimization, particularly micro-optimizations. Basically, optimization is fun. Thinking about application issues can be a lot less fun, particularly if it involves contact with those icky users. In the long run, however, it's application issues that will dominate. They may even dominate performance. Another ParcPlace conference attendee, whose name I didn't catch, commented that the biggest optimization for an operation is realizing that you don't need to do it at all.

That being said, it's nice if the operations you do need can run efficiently, and what follows are a few more tips for achieving this.

EXPLOITING SYSTEM LOOPHOLES

As we all know, in Smalltalk everything is an object, and all operations are accomplished by sending messages to objects. At least it looks that way. In fact, there are a number of situations where the system "cheats" to improve efficiency. Most of the time you don't need to worry about these situations, but it can be very helpful to know about them when you' re trying to optimize for performance.

INLINED MESSAGES

There is a small set of messages which are not implemented as message sends, but are instead "inlined." The compiler replaces the message send with code to perform the operation directly. While the exact set of messages replaced this way varies with the implementation, there are some which are almost always included.

Alan Knight is a consultant with The Object People. He can be reached at 613.225.8812, or by email as knight@acm.org.

Identity

One of the most important inlined messages is the == (double-equals) identity test. Equality testing is implemented by a message send, but the identity test is always a direct comparison of object pointers (you couldn't change it if you tried). This is the reason that things like IdentityDictionary and IdentitySet are faster than Dictionaries and Sets. You have to be careful however, that an identity test is really what you want. Usually you really want equality, and use identity as a substitute when you know they're the same for the objects in question. Identity tests on certain system classes are almost always safe:

SmallInteger:	myVar == 3
Symbol:	mode == #debug
UndefinedObject:	result == nil

Others are definitely not safe, and can get you in a lot of hard-to-debug trouble:

String:	mode == 'debug'
Numerics other than SmallInteger:	arg == 1.0

In particular, if you have a class where there is an equality operation different from identity, you probably want to think very carefully before using identity tests.

When using symbols, you should also be careful that you aren't paying the cost of the lookup in the conversion to symbol form. The expression aString asSymbol is doing a lookup in a very large hash table, and can have significant overhead if you're doing it many times.

Control structures and blocks

There are several basic control structures that are almost always inlined. This includes conditionals (ifTrue:, ifFalse:, etc.), while loops (whileTrue: etc.), Boolean operations that take a block (and:, or:) and indexed iteration (to:do:). These control structures are particularly important, because in addition to the message, the associated block is inlined as well. This can often make a much larger performance difference than just one message send.

BLOCKS

Blocks are a very important mechanism in Smalltalk, and an important feature that many static OO languages are missing. Since they are used to implement all of Smalltalk's control structures, they occur very frequently, and it's worthwhile to understand their performance characteristics. ParcPlace Smalltalk does the most optimization based on block types, categorizing them as follows:

Full blocks

These blocks can read and assign to temporary variables in the method context, and can contain a return. This requires transforming the method context into a real object (instead of a stack-based pseudo-object). For example, take the following (very inefficient) code fragment:

```
| aHugeString |
aHugeString := String new.
Smalltalk keys do: [:each | aHugeString := aHugeString, each.].
```

Copying blocks

These blocks can read temporary variables in the method context, but cannot assign to them or contain a return. It can read and/or assign to instance variables of self. The name comes from the fact that this can be implemented by copying the

Don't use isKindOf:, It's not just slow, it's the wrong thing to do.

appropriate values into the block, avoiding direct references to the method context in which the variables appear. The method context can therefore stay on the stack.

```
| aStream |
aStream := WriteStream on: String new.
Smalltalk keys do: [:each | aStream nextPutAll: each].
```

Clean blocks

These do not refer to variables except block arguments and globals. This is a very restrictive form, since it doesn't even allow references to self, or to instance variables. Because of the restrictions, this can be compiled as a simple function, which is much more efficient than the more general block forms.

```
| aStream |
aStream := WriteStream on: String new.
Smalltalk keys inject: aStream into: [:sum :next |
    sum nextPutAll: next].
```

Inlined blocks

These blocks have no restrictions on what they can do, but must be a literal block used as an argument to one of the inlined control structures described previously. Because of the way that they are used, the compiler knows enough to remove the block entirely, and directly embed the code within the method.

```
| keyStream aHugeString |
keyStream := ReadStream on: Smalltalk keys asArray.
aHugeString := String new.
[keyStream atEnd] whileFalse: [
    aHugeString := aHugeString , keyStream next].
```

Example block optimizations

These different block types are a mixed blessing. While they allow significant performance optimizations in some cases, they require some sophistication to understand. It's also hard to know exactly where they would be worthwhile, since block overhead is usually hard to spot in a profiler. One example of a performance optimization would be to re-implement detect: for SequenceableCollections. The default implementation is the inherited one from Collection:

```
detect: aBlock ifNone: exceptionBlock
    self do: [:each | (aBlock value: each) ifTrue: [^each]].
    ^exceptionBlock value
```

The ifTrue: block is inlined into the do: block, since ifTrue: always inlines literal blocks. This makes the do: block full, since it now contains a return. This can be quite inefficient, but there's no obvious way of getting around it. We need to return before we've finished the loop, since that's the essence of detect:. For general collections there isn't a very clean way to do this, but we know that SequenceableCollections can be addressed by index, and that to:do: inlines its argument. This allows us to write:

```
detect: aBlock ifNone: exceptionBlock
    1 to: self size do: [:i |
        | each |
        each := self at: i.
        (aBlock value: each) ifTrue: [^each]].
    ^exceptionBlock value.
```

Now the only "real" blocks are the ones passed in by the user, and the efficiency should be considerably improved. Because this small ugliness is hidden inside the implementation of a standard system method, it shouldn't affect the quality of our other code.

This is also a good example of a more general optimization technique, exploiting restrictions in subclasses. General classes like Collection need to provide very general methods. If we know that a subclass doesn't require the full generality, we can exploit that knowledge to provide a more efficient implementation. In this case, the knowledge we've used is that do: can be implemented just as efficiently using to:do: and at: for array-based sequenceable collections, and that all the sequenceable collections in the base image are array based.

This can get us in trouble if our assumption is violated. If we added non-array based implementations of sequenceable collections (e.g., self-balancing trees) at: could be a relatively expensive operation. This could easily make our "optimized" implementation less efficient than one based on do:. We would need to ensure that each type of collection used the appropriate detect: implementation.

Other implementations

One problem with block optimizations is that they're usually not portable. Digitalk's blocks are implemented quite differently than what is described previously, with fewer opportunities for optimization. While I am not as familiar with the details of Digitalk's blocks, they appear to have only the full and inlined varieties. They are also less general (they don't allow block locals, and block arguments are turned into variables in the method scope). I'm not sure, but these limitations may allow them to be implemented more efficiently than ParcPlace-style full blocks. In any case, there are fewer options in Smalltalk/V for these subtle optimizations.

IBM seems to do some very odd stuff with blocks. I say seems, because I haven't seen any of their implementation documented, and am guessing based on the code that is visible. IBM, like Digitalk, appears to only distinguish full and inlined blocks, but they also have a number of nonblock objects that can impersonate commonly used clean blocks. For example, the default sortBlock:

continued on page 32

JUANITA EWING

Exceptional power and control

In my last column, I discussed return values, and the use of specialized return objects. A return statement is only one mechanism that controls exit semantics and values. Another mechanism, an extremely powerful one, is exceptions. Exceptions provide flow control that cross and encompass methods.

The examples in this column are from the Smalltalk/V exception handling system. Objectworks\Smalltalk also has an exception handling system, so I will point out equivalent expressions during the discussion. We will also use the return object example from the last column as an example of how to add exceptions to an existing subsystem, including some architectural suggestions.

SIMPLE USE OF EXCEPTIONS

With an exception handling system, a developer can control how exceptional situations manifest themselves. In most Smalltalk systems, exceptions manifest themselves as errors, resulting in a walkback or error notifier. Exception handling gives developers the ability to control the manifestation of errors from low-level code, so they don't bubble up to the end user.

The basic premise of an exception handling system is simple: errors cause exceptions. Client code can ignore exceptions, which triggers the default action for the exception, or client code can handle the exception by performing a special action. Clients must designate which sections of code are protected from the default action of an exception.

To protect sections of code, the code must be placed in a block, and sent the message on:do:. The first argument to the message is the exception the developer wishes to handle. The second argument is the handler block, which is the code to be executed in case of an error. The handler block optionally has a block argument, which is the exception that was raised. In Objectworks\Smalltalk, the equivalent exception handling capability is invoked by sending the message handle:do: to an exception.

The following method isActive uses exception handling in a straight forward manner. The code makes sure that file errors do not interfere with the test for the existence of the info file, by placing the test in protected block. It invokes protected execution with the message on:do:. The argument to the on:do: message specifies the exception FileError. The handler block contains the special action that is executed if a FileError is raised during execution of the protected block. If there are file errors accessing the info file, we assume the file does not exist, and exit from the method with a return value of false. In this method, the handler block does not have an argument, which is the simpler form of the handler block.

```
isActive
    "Answer <true> if there is already an info file in the receiver's
      directory"

    ^[self infoFile exists]
        on: FileError
        do: [^false]
```

The on:do: statement in the isActive method handles the exception FileError and all of the exceptions derived from FileError. Unrelated exceptions are not handled in this statement. In the Macintosh implementation of Smalltalk/V, the FileError hierarchy looks like this:

```
FileError
    DirectoryNotFound
    EndOfFile
    FileDoesNotExist
    VolumeNotFound
```

These derived exceptions are specific kinds of file errors. In Smalltalk/V, exceptions are implemented as classes, so you can use standard browsing tools to examine and edit the exception hierarchy.

EXAMPLE DESCRIPTION

Our example from the last column used an operation that had several different return values, requiring the client to execute conditional code or perform a kind of case statement in order to use the result of the operation. We rearchitected the solution for our example, ending up with a specialized return object. The specialized return object could be queried to determine the success of the operation, and included more queries to determine if an exceptional condition had arisen.

Here is the description of our example operation:

- It might not succeed.
- The operation has a second chance of success—it can be retried with some input ignored.
- If the operation fails, it might be because of an internal error, or because an external function failed. For debugging purposes, it is desirable to distinguish between the two.
- Another effect of the operation is the creation of an OrderedCollection of strings containing result data from the operation.

The invocation of the operation using a specialized return object (simplified slightly from the previous column):

```
invokeOperation
    "Invoke the operationWithPoorInterface. Return a
      collection of strings if the operation succeeded. If it failed return
      an empty collection."
```

Juanita Ewing is a senior staff member of Digitalk, Inc. She has been a project leader for commercial object-oriented software projects, and is an expert in the design and implementation of object-oriented applications, frameworks, and systems. Previously, at Textronix Inc., she developed class libraries for the first commercial-quality Smalltalk 80 system. She can be reached via email at juanita@digitalk.com or by mail at Digitalk, Inc., 7585 SW Mohawk Drive, Tualatin, OR 97062.

```
| result |
result := self operationWithPoorInterface.
result wasSuccessful
    ifTrue: [self notifySuccess.
        ^result stringCollection].
result wasDataIgnored
    ifTrue: [self notifyDataIgnored.
        ^result stringCollection].
self notifyError: result errorMessage.
^OrderedCollection new
```

With the goal of simplifying the interface to the specialized return object, we can rewrite this invocation using exceptions. Although the initial version of the invocation does not have as much capability as the original version, we will improve on the exception version of the invocation as this article progresses. Here is the simple initial version:

```
invokeOperation
    "Invoke the operationWithPoorInterface. Return a
     collection of strings if the operation succeeded.
     Return an empty collection on failure."

    | result |
    [result := self operationWithPoorInterface]
        on: OperationWithPoorInterfaceError
        do: [: exception |
            self notifyError: exception errorMessage.
            ^OrderedCollection new].
    self notifySuccess.
    ^result stringCollection
```

In this version the operation is performed while protected from errors, using the on:do: message. If no errors occur, we execute the code after the on:do: message, which notifies the user of success and returns the result. If an error does occur, we notify the user of an error.

WHY IS THIS ARCHITECTURE BETTER?

The main difference between the example invocation with exceptions and without exceptions is the use of the on:do: message and the number and kind of messages sent to the specialized result object. The original invocation contained queries to the return object about errors. The new version does not contain queries about errors. The original specialized return object had information about two things: error conditions and operation results. With exception handling mechanisms, we can move the information about errors to the exception object. This partitioning of responsibilities results in more understandable and reusable code.

Though it is not so obvious by analyzing the client code, the developer has better control mechanisms with exception handling. The basic capabilities of the exception handling system allows the developer to elegantly handle errors generated at a low level. This is extremely important for complex operations. Behind the original implementation there was special purpose code containing specialized calls to low level operations that prevents low level errors from bubbling up to the user. The specialized invocations are eliminated by using exception handling.

HOW DO WE USE SPECIALIZED EXCEPTIONS?

Because exceptions in Smalltalk/V are implemented as classes, it is easy to extend the exception hierarchy using the same mechanisms used for extending the class hierarchy. If you have a need for specialized exceptions, then you should create an extension of the exception hierarchy. It is convenient to root all related exceptions at a single exception. This allows clients to write simple code to catch all related exceptions.

Most developers create a set of exceptions for each subsystem. This simplifies the interface between subsystems by providing a consistent and extensible way to pass error and notifications between subsystems.

Most systems have different exceptions for different kinds of errors because the client needs to distinguish between kinds of errors. When you are designing your hierarchy, for example, you might want to group resumable exceptions together. After analyzing our example operation, we decide to use an exception hierarchy like this:

```
Error
    OperationWithPoorInterfaceError
        PoorInterfaceExternalError
            PoorInterfaceFileError
            PoorInterfaceResourceError
        PoorInterfaceInternalError
            PoorInterfaceMissingInputDataError
            PoorInterfaceUncomputableError
            PoorInterfaceConflictingDataError
```

We want to distinguish between internal and external errors because the operation can be reattempted after an internal error. In this hierarchy, we make the distinction explicit by creating an exception hierarchy for each kind of error.

Why do we root our exception hierarchy at Error? One reason is that we want to inherit the appropriate behavior. One indicator of behavior is the default action of an exception. Here are the high level exceptions in the system, along with their default action:

```
Exception—open a walkback
    Error—open a walkback
    Notification—no action
        Warning—open a warning message dialog
```

The errors we generate from our example are serious problems, not just warnings or notifications. That makes Exception and Error potential derivation roots of our example exceptions because they have the appropriate default action: opening a walkback.

Of these two possibilities, we choose to derive our new exception from Error. We choose Error because it fits the standard way to catch all errors—an on:do: statement handling Error. The alternative is to catch all errors by handling Exception, but that combines catching errors and notifications. It is rare to want to treat notifications like errors!

EXTENDING EXCEPTIONS

In addition to creating new exceptions, the Smalltalk/V exception system also has the capability of extending exceptions by adding behavior or state. It is good practice to limit extensions to your own exception classes, so that your extensions do not collide with modifications made by the vendor.

Let's return to our invocation of the operationWithPoorInterface. The retry mechanism is convenient for allowing the end user to control this operation. Once we have determined the set of exceptions for our operation, we also want to implement a new message to determine if the operation can be retried. If the error is internal the end user is notified that he can retry the operation.

```
invokeOperation
    "Invoke the operationWithPoorInterface. Return a
     collection of strings if the operation succeeded. If it failed attempt
     a retry after user confirmation. Return an empty collection on
     failure."
    | result |
    [result := self operationWithPoorInterface]
        on: OperationWithPoorInterfaceError
        do: [: exception |
            (exception canRetryOperation and:
             [self canIgnoreData])
                ifTrue: [self notifyRetryPossible: exception
                errorMessage]
                ifFalse: [self notifyError: exception
                 errorMessage].
            ^OrderedCollection new].
    self notifySuccess.
    ^result stringCollection
```

The message sent to the exception to determine whether the operation can be retried, canRetryOperation, is a nonstandard message. It must be implemented by a specialized exception hierarchy.

We implement the message canRetryOperation at two different spots in our exception hierarchy. At the top, in the exception OperationWithPoorInterfaceError, we implement canRetryOperation to return false. For PoorInterfaceInternalError, we implement canRetryOperation to return true.

Developers can add state to exceptions, if necessary, by adding instance variables. The state inherited from Error includes an error message, but various other exceptions contain specialized information. For example, the exception MessageNotUnderstood has state for the message which is not understood. In our example exception hierarchy, we could add state to the conflicting data error, PoorInterfaceConflictingDataError, to describe which data are conflicting.

HOW ARE EXCEPTIONS GENERATED?

Our example showed us how to handle exceptions. We also need to know how to generate exceptions at the appropriate times. In our original example, the specialized return object contained error information.When we use exceptions, we need to replace code that stuffed error information into the specialized return object by code that raises exceptions instead. Let's examine a code fragment that used the specialized return object:

```
externalError := self externalOperation.
externalError >0
    ifTrue: [aPoorInterfaceResult errorCode: externalError.
        ^aPoorInterfaceResult].
```

Instead of sending messages to the return object, we need to rework this code fragment to raise an exception. The default way to raise an exception is to send the message signal or signal: to an exception. Our reworked code looks like this:

```
externalError := self externalOperation.
externalError >0
    ifTrue: [PoorInterfaceExternalError signal: (self
     errorMessage:externalError)].
```

The signal message raises an exception. The signal: message raises an exception accompanied by a descriptive message. Other exceptions have specialized instance creation message appropriate for their extended state.

From one error, we can create another kind of error. To do this, we handle the first error, and from the handler block raise another error. In this code fragment, we catch a file error, and raise a specialized file error:

```
[fileStream := self createTemporaryFile]
    on: FileError
    do: [:exception |
        PoorInterfaceFileError signal: exception message]
```

FINER CONTROL

There are a variety of ways to exit from an exception handler, each providing a different form of finer control. Exit mechanisms include resume, return, pass and retry. Some or all of these mechanisms are extremely useful with multiple exception handlers, but can also be useful with a single exception handler. All of these mechanisms are invoked by sending messages to the exception inside the exception block. Of these mechanisms, we will discuss retry and resume in detail.

The retry mechanism is used to re-evaluate the protected block, the receiver of the on:do: message. It is invoked by sending the exception the message retry. There is a variation of retry that allows an alternate block of code to be evaluated. It is invoked with the message retryUsing: and takes the alternate block as it's argument. In Objectworks\Smalltalk, the retry mechanism is invoked with the message restart.

We again come back to our specialized return object example. Our original example included information describing whether the operation had been attempted again. The client had no control over the re-attempt. With exceptions, we can improve the invocation of the operation by moving the retry control to client. With the retry mechanism incorporated, the invocation looks like this:

```
invokeOperation
"Invoke the operationWithPoorInterface. Return a
 collection of strings if the operation succeeded. If it failed attempt a
 retry after user confirmation. Return an empty collection on failure."
        | result |
        [result := self operationWithPoorInterface]
            on: OperationWithPoorInterfaceError
            do: [: exception |
                (exception canRetryOperation and: [self
                 canIgnoreData])
                    ifTrue: [self confirmIgnoreData
                        ifTrue: [self ignoreData.
                            exception retry]].
                "Can't retry"
                self notifyError: exception errorMessage.
                ^OrderedCollection new].
```

```
        self notifySuccess.
        ^result stringCollection
```

If queries indicate data can be ignored, then the operation is retried by ending the message retry to the exception. We continue to make use of extensions to query the exception.

Here is an example using the retry mechanism from the Macintosh version of Smalltalk/V. One of the classes that manages memory, AbstractMemoryHeapPolicy, has a method that is used to allocate heap memory. If the allocation fails, indicated by the exception MacNotEnoughMemory, then a low memory action is performed to attempt to recover space and the allocation is retried.

```
AbstractMemoryHeapPolicy
do: aBlock requiringHeapBytes: estimatedHeapBytes
"Evaluate <aBlock> after verifying that there is enough room on the
 heap to allocate <estimatedHeapBytes>. Perform
 lowHeapMemoryAction: if there isn't enough room.
        Simplified for example."

    ^(self roomOnHeapFor: estimatedHeapBytes)
        ifTrue:
            [aBlock
                on: MacNotEnoughMemory
            do:
                [:ex |
                (self lowHeapMemoryAction: estimatedHeapBytes)
                    ifTrue: [ex retry]]]
```

The other control mechanism I want to spend some time discussing is resume. Resume is a control mechanism that tells the exception handler to "keep going." Only resumable exceptions can be resumed.

```
MainWindow
close
    "Time to close the receiver. Check with the model, don't
    close if it doesn't want to."

    | allowClose |
    allowClose := true.
    [self triggerEvent: #aboutToClose]
        on: VetoAction
        do: [:ex | allowClose := false. ex resume].
    allowClose
        ifTrue:
            [self closeWindow]
```

In the close method, the aboutToClose event is sent to all objects that have registered an interest in the event. If any of the registered objects want to disallow closing, they signal a veto by raising the VetoAction exception. But, the processing shouldn't stop because of a veto. Each registered object must receive the aboutToClose event. The code is designed to handle this requirement: it notes the veto by setting the allowClose Boolean to false, and proceeds to finishing informing registered objects about the intent to close by resuming the protected block. After the protected block is completely executed, informing the entire set of registered objects of the intent to close, the window is closed if no object has vetoed the close.

WHICH ERRORS SHOULD YOU CATCH?

When handling errors, a good rule of thumb is to handle the most specific error that is appropriate. Specific handling is usually better than general handling, especially during development.

A common mistake is to write code that inappropriately handles the exception Error. More than one developer has been mystified by the cause of an exception, only to discover that their code catches all errors, including MessageNotUnderstood, a subclass of Error. In this case, generalized error handling covered up a coding mistake.

ENSURED EXECUTION

Another mechanism, built on exceptions, is the ability to ensure execution of some code. This mechanism requires placing protected code in a block, and the code whose execution must be guaranteed in another block. Ensured execution will execute the ensured code no matter what happens, even if a return expression or an error terminates the protected block early.

This mechanism is particularly useful in cases that must reset state or that must be protected against inconsistencies. For example, this mechanism can ensure that a file will be closed after reading data from it. Smalltalk/V uses the message ensure:, which is sent to a block containing protected code and has the guarantee block as its argument. The Objectworks\Smalltalk equivalent is valueNowOrOnUnwindDo:.

This example is from the Macintosh version of Smalltalk/V. The method fill:withColor: uses ensured execution to make sure the background color is reset to its previous value. The background color will be reset from the guarantee block, even if the erase operation from the protected block signals an error.

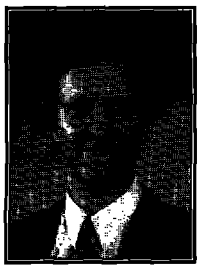
```
GraphicsTool
fill: geometricObject
withColor: fillColor
"Fill the inside of a <geometricObject> with the given <fillColor>. The
 location of the receiver is not affected."

    | backgroundColor |
    backgroundColor := self backColor.
    [self backColor: fillColor.
    geometricObject eraseOn: self]
        ensure: [self backColor: backgroundColor]
```

Another related mechanism is one that guarantees execution of some code in case of an error. This code is executed only in the case of abnormal termination, such as with an error. The Smalltalk/V message to invoke this mechanism is ifCurtailed:. The Objectworks\Smalltalk equivalent is onUnwindDo:.

CONCLUSION

Exception handling is a powerful mechanism for controlling errors and notifications. Even simple applications can benefit from ensured execution and handling predefined exceptions. Complex applications can benefit from specialized exceptions. Each subsystem in the application should define specialized exceptions that are part of the public interface of that subsystem.♀



MARK LORENZ

When the worst happens

This article could be entitled "Making a smooth recovery." It deals with those times when lightning hits the building, your dog trips over the power cord, your kid plays with that interesting button on the power strip, or (as recently happened for me) you jump on a beta and/or newly released platform. In other words, when your simulated world of objects comes crashing down around your ankles, what's the best way to pick up the pieces?

RECOVERING WITH ENVY

ENVY/Developer is a multi-user development environment for Smalltalk. It is used with VisualWorks from ParcPlace, Smalltalk/V from Digitalk, and VisualAge from IBM. ENVY keeps most changes in a server-based database, generally in a file named MANAGER.DAT. Changes come in the form of *editions* and *versions*. An edition is a component that is still being worked on. A version is an edition that has been released for public consumption and can no longer be directly changed (although another edition of it may be created).
Recovering with ENVY is a relatively painless multistep process.

1. **Make image consistent:** This action is unfortunately placed in different places in different versions of ENVY. It is however always cascaded as a submenu under *System*.

- In VisualWorks, it is found on the popup menu for the ApplicationManager applications pane.
- In Smalltalk/V, it is found on the Transcript's *ENVY* menu.
- In VisualAge, it is found on the Transcript's *Smalltalk tools* menu.

The make image consistent action looks for inconsistencies between the editions on the server and those in the image. Results appear in the Transcript. Warnings and errors should be dealt with as documented in the manual. Informational messages can be deleted. Depending on the version of ENVY, you may also get one or more pop-up windows detailing the inconsistencies found and allowing *Load alternative* actions to resolve them.

Mark Lorenz is founder and president of Hatteras Software, Inc., a company that specializes in helping other companies use object technology effectively. He welcomes questions and comments via e-mail at 71214.3120@compuserve.com or phonemail at 919.851.0993.

2. **Available classes:** This action is available from the *Classes* menu in the Application Manager.* It lists classes that are not loaded in the image, but are part of another edition of the application. Classes can be missing if you crashed after creating a class, but before saving your image. Check this for each application you were working on that may have missing classes and reload them into your image.

3. **More recent editions:** This action is also available from the *Classes* menu in the Application Manager. It lists any editions that have a time stamp later than the one loaded for any of the application's classes. Check this for each application you were working on before you crashed, loading later editions as needed.

At this point, you should be recovered from your crash. This is a good time to save your image.

If your image is corrupt and cannot be run, you should start from a fresh image and load from your configuration map, following the steps above to recover your work in progress. If you don't have a configuration map, then you'll have to load application versions by hand... dealing with configuration maps is a subject for another entire article.

RECOVERING WITH TEAM/V

Team/V is a multi-user development environment for Smalltalk/V. It uses the PVCS library system for storage of packages in disk files of the form <package name>.PKV or the file system to store flat files of the form <package name>.PKG.

Recovery using Team/V is also a relatively painless process.

1. **Migrate packages:** Any packages that have been committed will have need to be "migrated" to the latest version of the code in the shared library. This is done from the packages menu.

2. **Recovery from the change log:** Changes that have not been committed will not be in the library, so you will have to retrieve them from the change log. This is similar to the section on "vanilla" Smalltalk for Smalltalk/V, with the difference that Team/V information will also be logged. You will only file in the code changes and not the Team/V logged messages.

3. **Reassignment to packages:** Unfortunately, the changes you install from the change log that are for new classes and methods will not be automatically assigned to a package. Fortunately, you can do this through direct action drag and drop from the temporary "stream" package to the desired package(s).

4. **Cleanup:** Delete any temporary packages created for change log installs. Update any package browsers from the packages menu.

RECOVERING WITH "VANILLA" SMALLTALK

The Smalltalk environment itself has some safeguards built into it. The different vendors' versions work somewhat differently.

Smalltalk/V

Smalltalk/V uses a file called CHANGE.LOG that has modifications to the environment logged to it in such a way that you can recover from it. The format of the changes is called *chunk* format. Chunks are groupings of text delineated by exclamation points ("!").

* Again, the different versions allow access to the Application Manager from different places. You can bring up the Application Manager from the Manage applications option of the ENVY (VisualWorks or Smalltalk/V) or Smalltalk tools (VisualAge) menus.

Project Practicalities

The following example taken from one of my change logs illustrates the types of information kept in this file.

```
"define class"

HomBaseBrowser subclass: #HomModelBrowser
    instanceVariableNames: "
    classVariableNames: "
    poolDictionaries: "!

"evaluate"
HomModelBrowser example!

!HomTopPane class methods!
fileMenu
        "Public - return the File menu for my browsers to use"
    "MEL 3/1/93 ©Hatteras Software, Inc. 1993. All rights reserved."
    ...
    ^menu! !

"evaluate"
HomTopPane class removeSelector: #smalltalkMenu!

!HomTopPane methods !
buildMenuBar
        "Private - Create the menus that make up the menu bar."
        "MEL 3/1/93 ©Hatteras Software, Inc. 1993. All rights reserved."
    ...! !
```

To recover: From Smalltalk, open the file with your changes in it. You may have to force the entire file to be read into the workspace browser.

- Place the cursor at the bottom of the file.
- Bring up the Find dialog. Enter "image," choose "backwards," and press Enter.
- Page forward, deleting any "evaluation" lines you don't want to execute again.
- Place the cursor at the beginning of the line after the "image saved" line.
- Select the text from there to the end of the file.
- File the selected text into the image.

VisualWorks

VisualWorks has a different way of dealing with change control. As you make changes, they are put into a change log named VISUAL.CHA. VisualWorks provides a browser that separates the types of changes that are logged, allowing better viewing in some ways. It does not allow you to view only the changes since the last image save, however.

Installing the changes is usually done via the browser's menu options, instead of through selections in the change log itself using a workspace.

SUMMARY

We have taken a look at disaster recovery when using Smalltalk. We've seen that, although it takes some effort to recover, Smalltalk allows for relative safety when using it to develop applications. Groupware for Smalltalk makes the situation even easier to handle. Of course, you can safeguard yourself even further by frequent image saves, commits/versioning to the repositories, and regular archival backup and off-site storage. ♀

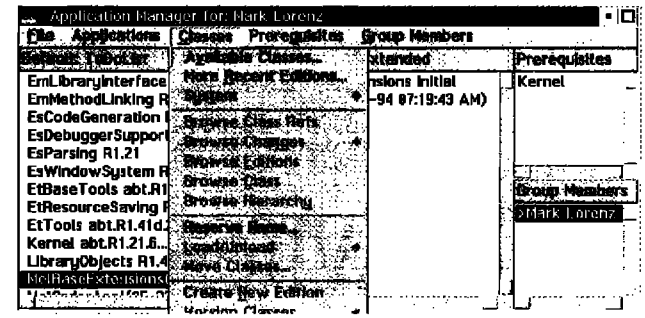


Figure 1. ENVY Application Manager menu.

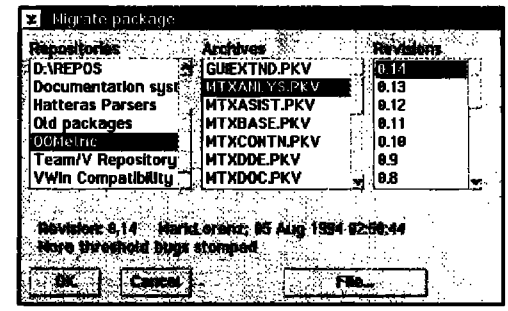


Figure 2. Team/V migrate applications dialog.

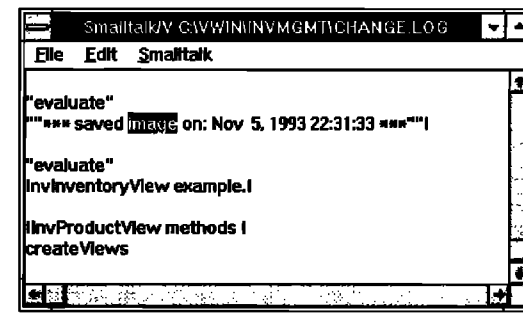


Figure 3. An example Smalltalk change log.

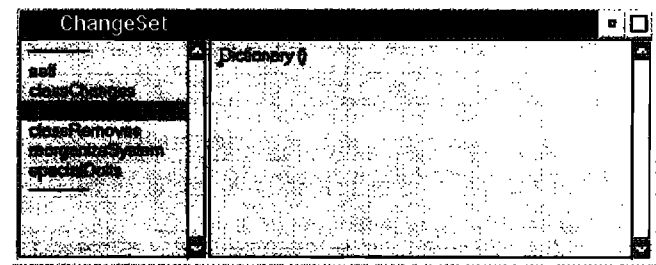


Figure 4. VisualWorks changes browser.

References

1. ENVY/DEVELOPER USER MANUAL, Object Technology International, 1993.
2. TEAM/V USER MANUAL, Digitalk Incorporated, 1993.
3. OBJECTWORKS SMALLTALK USER'S GUIDE, ParcPlace Systems Inc., 1992.

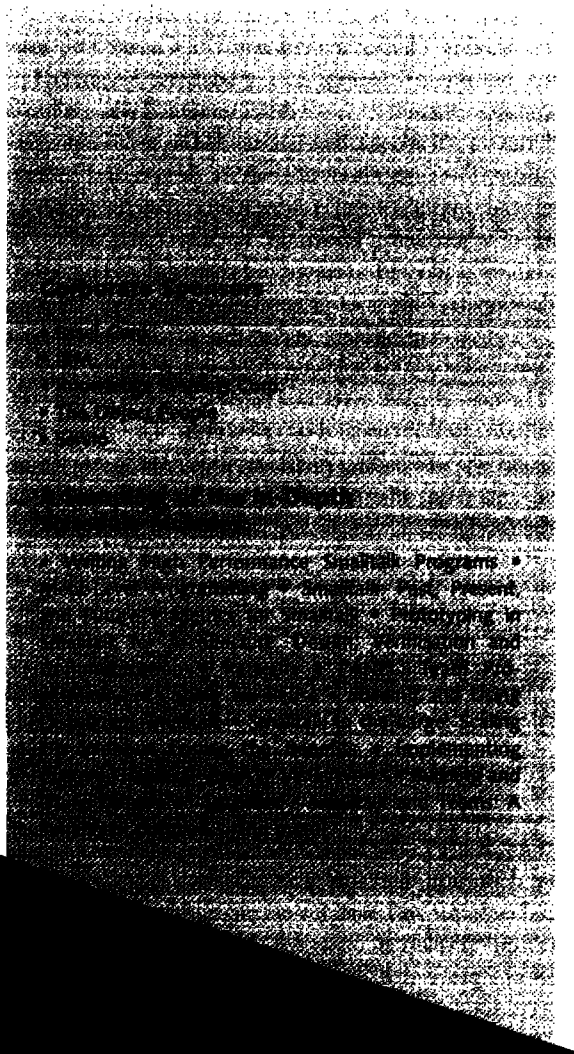
February 21–24, 1995

Omni Park Central

New York, NY

SMALLTALK SOLUTIONS 95

Where All the Talk is Smalltalk



Finally, a vendor-independent conference dedicated to all Smalltalk users. Focusing on the practical application of Smalltalk in its dialects, **Smalltalk Solutions '95** is an opportunity for the entire Smalltalk community to network, share innovative strategies and programming tips, and stay up-to-date on the latest tools and techniques.

Learn from the Smalltalk Experts

The educational program has been designed in conjunction with the Technical Conference Chair John Pugh, editor of *The Smalltalk Report*. The 4-day conference offers over 30 intensive classes ranging from beginner to advanced, all taught by experienced and well-respected Smalltalk experts.

You'll come away with new insights on language advances, usage tips, project management, analysis and design techniques, and insightful, practical applications. Specific class tracks focus on Team Programming, Analysis and Design, User Experiences/Case Studies, Technical Features, and Management Issues.

The latest Smalltalk products will be displayed in the **Smalltalk Solutions '95** Exhibit Hall, where you'll have a chance to demo the leading Smalltalk products, and receive an up-close, hands-on comparison. Don't miss this chance to see Smalltalk in action.

Smalltalk Solutions '95 is presented by SIGS Conferences, sponsor of over 7 conferences world-wide, including **Object Expo, Object Expo Europe,** and **C++ World.**

SIGS CONFERENCES

For information on attending **Smalltalk Solutions '95**, please contact the SIGS Conferences Registrar:

PHONE: 212.242.7515

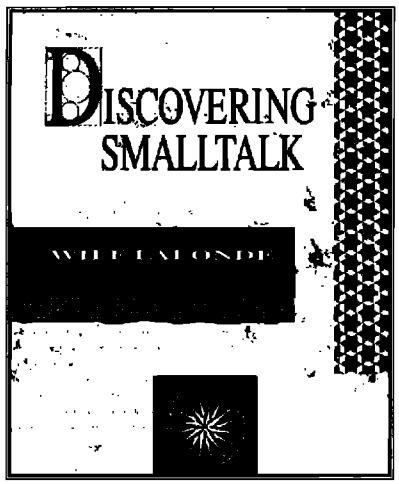
FAX: 212.242.7578

email: sigsconf@ix.netcom.com

DON'T MISS THIS UNIQUE SMALLTALK EVENT!

Book Review

Discovering Smalltalk



Reviewed by
Mary Dunn

Wilf LaLonde
Benjamin Cummings
Redwood City, CA
ISBN 0-8053-2720-7
1994

Wilf LaLonde is an author and educator whose name will be familiar to many readers of THE SMALLTALK REPORT. His latest book, DISCOVERING SMALLTALK, aims to provide an introduction to the Smalltalk language, environment, and class library, and to the major concepts of object oriented development. His educational philosophy is that learning is best done through experimentation and discovery. Hence, the book is organized as a process of discovery, plunging first into examples of actually doing some simple things in the Smalltalk environment, and then proceeding from basic to complex topics. The basis for the text is Smalltalk/V for Windows; the final chapter introduces user interface development using Object Share's WindowBuilder in its examples.

The intended audience is "those who have never programmed before," as well as those who know other programming languages. It is designed for use in a first course in object oriented programming. In the preface, LaLonde discusses his academic curriculum and where this book is used in sufficient detail to be useful as a guide to educators. Outside of the classroom, I found the book to be a good vehicle for an information systems professional learning Smalltalk through self-study. It would be especially effective following some introductory reading on O-O, such as David Taylor's OBJECT ORIENTATION: A MANAGER'S GUIDE.

I have a background in corporate information systems rather than in the academic sector. I am currently managing a project to introduce object-oriented technology to our company, which includes a first project using Smalltalk/V and WindowBuilder. I have done some research into object orientation and taken a course in Smalltalk and O-O analysis/design but had not used Smalltalk prior to reading this book.

The structure of the book provides a sound framework for learning: Each chapter explains what it will cover and why and ends with several sections that reinforce learning. Throughout the text, important facts and ideas are enclosed in bordered boxes to capture the reader's attention. I found the index a bit sparse, but I could locate most topics using the table of contents.

Discovering Smalltalk begins with a brief introduction to computers, Windows, and Smalltalk. The coverage of basic computer, mouse, and Windows concepts seemed likely to be too little if the reader needed it and in the way if not. (I suspect the latter would be the more typical case.) We learn about Smalltalk messages, receivers, and selectors through simple text manipulation and calculations in the Workspace. Browsers are introduced by using them, and we move on to methods, classes, and debuggers. From here on, we see more examples of code and discussion of how Smalltalk works. The concept of inheritance and class hierarchies follows. Collections merit their own chapter, introducing the collection classes and how to create and process them.

At this point we take a different tack to consider evolutionary analysis and design techniques (primarily Class-Responsibility-Helper analysis and Use Cases). This discussion points out the interrelationship of analysis, design, and programming inherent in O-O development. I was glad to see this included in a textbook that is intended to teach a programming language. The integration of analysis, design, and coding has significant implications for information systems professionals, many of whom have "grown up" as either "analysts" or "programmers" and need to round out their skills to be effective O-O developers. This brief presentation of analysis/design skills and iterative development makes this a richer book, especially for the reader who is learning through self-study.

The next chapter gives us a fairly extensive look at the Smalltalk class library and its operations. The author has us try different ways of coding methods, thus reinforcing the experiment-and-refine method of development while introducing us to the fundamental classes and methods in Smalltalk. Suggestions for further exploration are made so that we gain confidence at learning through browsing. The basic structure and object interaction in the user interface of a Windows application are introduced very briefly in the final chapter.

The "discovery" method requires tolerance for incomplete understanding early in the process of learning, which may be uncomfortable, especially for the reader learning through self-study. The author helps us through reminders of where we'd seen a concept before and gives assurances that we'd revisit it later on. Early in the book, this reader would have appreciated more definition than was provided when topics were first introduced (selector and part, for example). However, I must admit that I did develop an experiential understanding of the concepts as I proceeded. Throughout the book, LaLonde offers observations and hints based on his experience; these are especially valuable as the topics become more advanced.

The integration of doing with learning is notably effective. If reading fails to convey an idea, then the example that faithfully follows usually succeeds. I was grateful for the extensive use of figures showing Smalltalk transcripts, browsers, code examples, etc., since it allowed me to make progress in the book even when I was not able to be hands-on with Smalltalk while

continued on page 32

Objekt-orientiertes Programmieren

OOP '95

MÜNCHEN

Don't Miss Germany's Most Attended Object Technology & C++ Conference!

Moving Forward with Object Technology

Here's What Prominent German Publications Said About OOP'94...

*"**OOP '94**: A lot of new exhibitors with new products..."*

—*Computerwoche*, Nr. 7, February 18, 1994

"For companies who want to give further education to their employees by participating in a conference or who are making important buying decisions...it is the only alternative."

—*iX Magazin*, March, 1994

"The most important thing for the attendees was the quantity and the quality of the talks and the seminars...It was a positive experience in contrast to mass-events like Cebit... For software developers it is recommended to visit this conference."

—*mc Magazin fur Computerpraxis*; Jan. 1994

FOR INFORMATION ON EXHIBITING OR ATTENDING OOP'95 FEATURING C++ World CONTACT:

(In the USA) SIGS Conferences, Inc.....v. 212.242.7515...f. 212.242.7578

(In Germany) SIGS Conferences GmbH.....v. 089.957.9517...f. 089.957.9125

Sponsored by:

Journal of Object-Oriented Programming

C++ Report

OBJEKTspektrum

Object Magazine

iX Multiuser Multitasking Magazin

Presented by:

SIGS Conferences

continued from page 15

```
        interfaces do: [:aTransportInterface | aTransportInterface close]].
    interfaces := nil.

stopListening
    | aProcess |
    listenerProcess notNil
        ifTrue:
            [aProcess := listenerProcess.
            listenerProcess := nil.
            aProcess terminate].

`initialize-release'

close
    super close.
    self stopListening.
    self closeInterfaces.

`client access'

addClient: aTransportInterface
    self interfaces add: aTransportInterface.
    aTransportInterface addDependent: self.

removeClient: aTransportInterface
    self interfaces remove: aTransportInterface ifAbsent: ["Just don't
care"].
    aTransportInterface removeDependent: self.

`message processing'

listenerProcessLoop: aPortNumber
    | childSocket newClient |
    transport := TransportInterface newAtPort: aPortNumber.
    [transport notNil] whileTrue: [
        childSocket := transport socket accept.
        childSocket isNil
            ifFalse:
                [newClient := TransportInterface newOn: childSocket.
                self addClient: newClient]]

`updating'

update: aSymbol with: anObject from: aSender
    aSymbol == #closedSocket
        ifTrue: [self removeClient: aSender]
        ifFalse: [super update: aSymbol with: anObject from: aSender].

SmalltalkServer class methods

`instance creation'

newAtPort: aPortNumber
    | aServer |
    aServer := self new.
    ^aServer interfaces: OrderedCollection new;
        listenerProcess:
            ([aServer listenerProcessLoop: aPortNumber]
            forkAt:Processor activePriority - 1);
            yourself.
```

Book Review

continued from page 28

reading. The book accompanied me to the porch when pleasant weather beckoned and to the dentist's waiting room when necessity dictated. Such portability is no small benefit to one learning on one's own time.

Code examples did get lengthy, as LaLonde had forewarned in the preface. However, through the numerous and fully realized examples, we become familiar with Smalltalk code almost subconsciously as examples build from chapter to chapter. In addition, the examples will serve as models after one has finished the book. The technique provides solidification and extension of understanding over the course of the book. In fact, the flow of the book in itself constitutes a demonstration of O-O's iterative style of development.

Confession being good for the soul, I must admit that I actually did not do the exercises at the end of the chapters. The good news for potential self-study readers like me is that the book worked anyway. I feel that I have landed firmly on the shores of Smalltalk and can p roceed inland with the mind-map provided by DISCOVERING SMALLTALK. ♀

The best of comp.lang.smalltalk

continued from page 20

```
[:a :b | a <= b]
```

appears to be transformed by the compiler into an object with the method

```
value: a value: b
    ^ a <= b
```

I'm not sure exactly what the performance benefits of this approach are, but it makes block optimizations even trickier, since you need to know which particular blocks are supported in this way to optimize.

A HAPPY ENDING

There are many more areas of performance optimization that I have not yet even begun to talk about, so there may be more forthcoming on this topic in the future, including arithmetic and graphics operations, as well as some well-known performance hits (e.g., Don't use isKindOf:. It's not just slow, it's the wrong thing to do). For the moment, I'm way past deadline and running out of space. I opened this two-part series with a quote from **Bill Punch (punch@cps.msu.edu)** who was looking to optimize some Smalltalk code. I'm happy to say that the results were satisfactory:

> **Using the acquired wisdom of the net, we have re-tuned our application with some pretty wonderful results. There are lots of ways to show this, but based on my home machine (NeXT, 32Mb, ParcPlace VW) we showed the following improvements (same test case shown):**
>
> | | |
> |---|---|
> | **Original June code:** | **42 seconds** |
> | **1st speed fix:** | **27 seconds** |
> | **2nd speed fix :** | **20 seconds** |
> | **3rd speed fix :** | **15 seconds** |
>
> **All this for three to four hours of effort, most spent trying to figure out the profiler and what it was telling us. Same code on our Sparc 20/50 (32 Mb, ParcPlace VW) runs in under 3.5 seconds with some copious output. In particular, we are now showing that TextCollector>>show: is taking something like 25% of the run time. That is, our code is running in comparable time to how fast VW can pump out the results. We're pretty happy with that!**
>
> **How did we do that? The main net advice is the use of the Profiler from the Advanced Programmers Kit (APOK). We hadn't used it much before but will from now on. We found a number of "holes" in our code that we fixed.**
> **The other oft offered advice is on "growing" collections. We found some overhead in that as well and remedied it.**

THE USUAL BUGS

One of the problems with taking information off the net is its short lifespan relative to the delays involved in publication. By the time the information on the Smalltalk standardization mailing list was published (THE SMALLTALK REPORT, July–August 1994) it was already out of date. Requests for subscriptions, which should be of the form:

```
subscribe x3j20 your-email-address
```

should now be sent to listserv@qks.com, and submissions to the mailing list now go to x3j20@qks.com. Problems with the list should be addressed to postmaster@qks.com. ♀